COORDINATED HIGHWAYS ACTION RESPONSE TEAM
STATE HIGHWAY ADMINISTRATION

R1B1 High Level Design

Contract DBM-9713-NMS
TSR # 9901961
Document # M361-DS-001R0

June 25, 1999
By
Computer Sciences Corporation and PB Farradyne Inc

Table of Contents

1 - Introduction

1.1 Purpose

This document describes the high level design of the Chart II software for Release 1, Build 1. This design is driven by the Release 1, Build 1 requirements as stated in document CHARTII-RS-001-00, "***CHART II System Requirements Specification For Release 1 Build 1***."

1.2 Objectives

The main objective of this design is to provide software developers with a framework in which to provide detailed design and implementation of the software components used to satisfy the requirements of Release 1, Build 1 of the Chart II system.

This design also serves to provide documentation to those outside of the software development community to show how the requirements are being accounted for in the software design.

1.3 Scope

This design is limited to Release 1, Build 1 of the Chart II system and the requirements as stated in the aforementioned requirements document.

1.4 Acronyms

The following acronyms appear throughout this document:

CORBA	Common Object Request Broker Architecture
DMS	Dynamic Message Sign
FMS	Field Management Station
GUI	Graphical User Interface
IDL	Interface Definition Language
ORB	Object Request Broker
UML	Unified Modeling Language

1.5 Design Process

Object oriented analysis and design techniques were used in creating this design. As such, much of the design is documented using diagrams that conform to the Unified Modeling Language (UML), a de facto standard for diagramming object oriented designs.

In addition to being object oriented, this design incorporates distributed object techniques, which allow for great flexibility and scalability of the system. In a distributed object system, objects can be deployed in servers throughout the network. This design addresses the partitioning of object types into specific server applications for this release.

The design process is very iterative, with each step possibly causing changes to previous steps. Listed below is the process that was used to create the work products contained in this document:

- The team first created a use case diagram to reflect the requirements listed in the requirements diagram. Each requirement was mapped to at least one use case. The mapping was documented in the DOORS requirements tool.
- A straw man class diagram was created with major entities evident in the use cases being listed as possible classes in the system. High level relationships between the classes were discovered and documented on the class diagram.
- A sequence diagram was created for each use case, showing how the classes on the class diagram would be used to perform the use case. This often involved changes to the class diagram, such as adding classes, moving responsibilities between classes, or adding operations to a class. Sometimes the changes affected other sequence diagrams as well.
- After the process of creating sequence diagrams and associated changes to the class diagram, internal reviews were used to resolve remaining issues.
- To enable the design to be manageable when taken into the detailed design phase, the design was broken into packages, grouping classes with a high amount of dependency together. This partitioning is documented on a package diagram. During this partitioning it was also decided which interfaces were needed to be used by other packages, thus defining the CORBA objects in the system.
- Using the class diagram as well as the sequence diagrams, the IDL for each of the CORBA objects was specified.
- A deployment diagram was created to show which applications would be responsible for each CORBA object.
- A second deployment diagram was created to show where each application would be deployed on the network.

1.6 Design Tools

The system requirements are stored in a tool named DOORS in a project named Chart II under a formal document named R1B1. Within the R1B1 document, an attribute was added by the design team to specify the use case(s) that map to each requirement.

The work products contained within this design (with the exception of the IDL) are extracted from the COOL:JEX design tool. Within this tool, the design is contained in the Chart II project, Release 1 configuration, Analysis phase, system version Core Classes 2.

The IDL was created through referring to the Class, sequence, and package diagrams contained within this document. A text editor (textpad32) was used for the actual creation of the IDL files.

1.7 Work Products

This design contains the following work products:

- A UML Use Case diagram, showing the different uses the system provides to the end user.
- A UML Class diagram, showing the high level software objects which will allow the system to accommodate the uses of the system described in the Use Case diagram.
- UML Sequence diagrams, one for each use case, showing how the classes interact to accomplish a use of the system.
- A UML Package diagram, showing how the classes are broken up into manageable software packages.
- A UML Deployment diagram, showing which servers will serve each class of objects. While this is not the typical use of a deployment diagram, the team found this diagram to meet their needs for describing this aspect of a distributed system.
- A UML Deployment diagram, showing where servers and GUIs will be deployed.
- OMG IDL for the distributed objects that will exist in the system.

2 - Software Architecture

The architecture used in Release 1, Build 1 is not limited to meeting the requirements for this single build and instead is designed as the building block for the entire Chart II system. Driven by the requirements of the Chart II system and taking into account guidelines for a national ITS architecture, the Chart II system is designed as a distributed object system utilizing the Common Object Request Broker Architecture (CORBA) as the base architecture. The remainder of this section discusses how CORBA is to be applied as the Chart II software architecture.

2.1 CORBA

CORBA is an architecture specified by the Object Management Group (OMG) for distributed object oriented systems. The CORBA specification provides a language and platform independent way for object oriented client/server applications to interact. The CORBA specification includes an Object Request Broker (ORB) which is the middleware used to allow client/server relationships between objects. Using a vendor's implementation of an OMG ORB, software applications can transparently interact with software objects anywhere on the network without the application having to know the details of the network communications.

Interfaces to objects deployed in a CORBA system are specified using OMG Interface Definition Language (IDL). Applications written in a variety of languages or deployed on a variety of computing platforms can use the IDL to interact with the object, regardless of the language or computing platform used to implement the object.

Included in this design document is the IDL for CORBA objects that provide the functionality for Release 1, Build 1 of the Chart II system.

2.2 CORBA Services

Included in the OMG CORBA specification are specifications for application servers that provide basic functionality that is commonly needed by distributed object systems. While there are specifications for many such services, many services have not yet been implemented. Of the CORBA Services that are available, the CORBA Event Service and CORBA Trading Service are utilized in the CHART II system. A description of each of these services follows.

2.2.1 CORBA Event Service

The CORBA Event Service provides for a way to provide data updates within the system in a loosely coupled fashion. This loose coupling allows applications with data to share to pass the information via the event service without needing to have knowledge of others that are consuming the data.

Data passed through the event service is done using event channels. Many different types of events may be passed on a single event channel. Interested parties may become consumers on a given event channel and receive all events passed on the channel.

The CHART II system makes use of multiple event channels to allow event consumers to be more selective about the type of events they receive. Also, event channels of the same type may

exist in multiple regions, allowing the CHART II system to be expandable and multi-regional. Event channels used in the CHART II system are published in the CORBA trading service to allow others to select which events they wish to consume.

2.2.2 CORBA Trading Service

The CORBA Trading Service is an online database of objects that exist in a distributed object system. Servers which have services to offer publish their objects in the trading service. Applications that wish to use the services provided by a server can query the Trading Service to find objects based on their type or attributes.

CORBA Trading Services can be linked together into a federation. Queries done on single Trading Service can be made to cascade to all linked Trading Services as well. This feature allows Trading Services serving single regions to be linked together, providing seamless access to all objects in the system.

The CHART II System utilizes the CORBA Trading Service to allow the GUI to discover objects in the system that it allows the user to interact with. Using the linking capabilities of the Trading Service, the CHART II system can be distributed to multiple districts with the GUI still able to provide a unified view of the system to the users.

2.4 Chart II Application Services

Custom application servers built for the CHART II system are used to house the CORBA objects that provide the functionality of the system. These application servers provide services to the objects contained within them, such as database connectivity, event channel management, and object publication. The design of each of these objects and their interaction is addressed in following sections of this document.

3 – Use Cases

The following diagram shows the possible uses of the Chart II Release 1, Build 1 system. These uses of the system are derived directly from the Release 1, Build 1 requirements document.

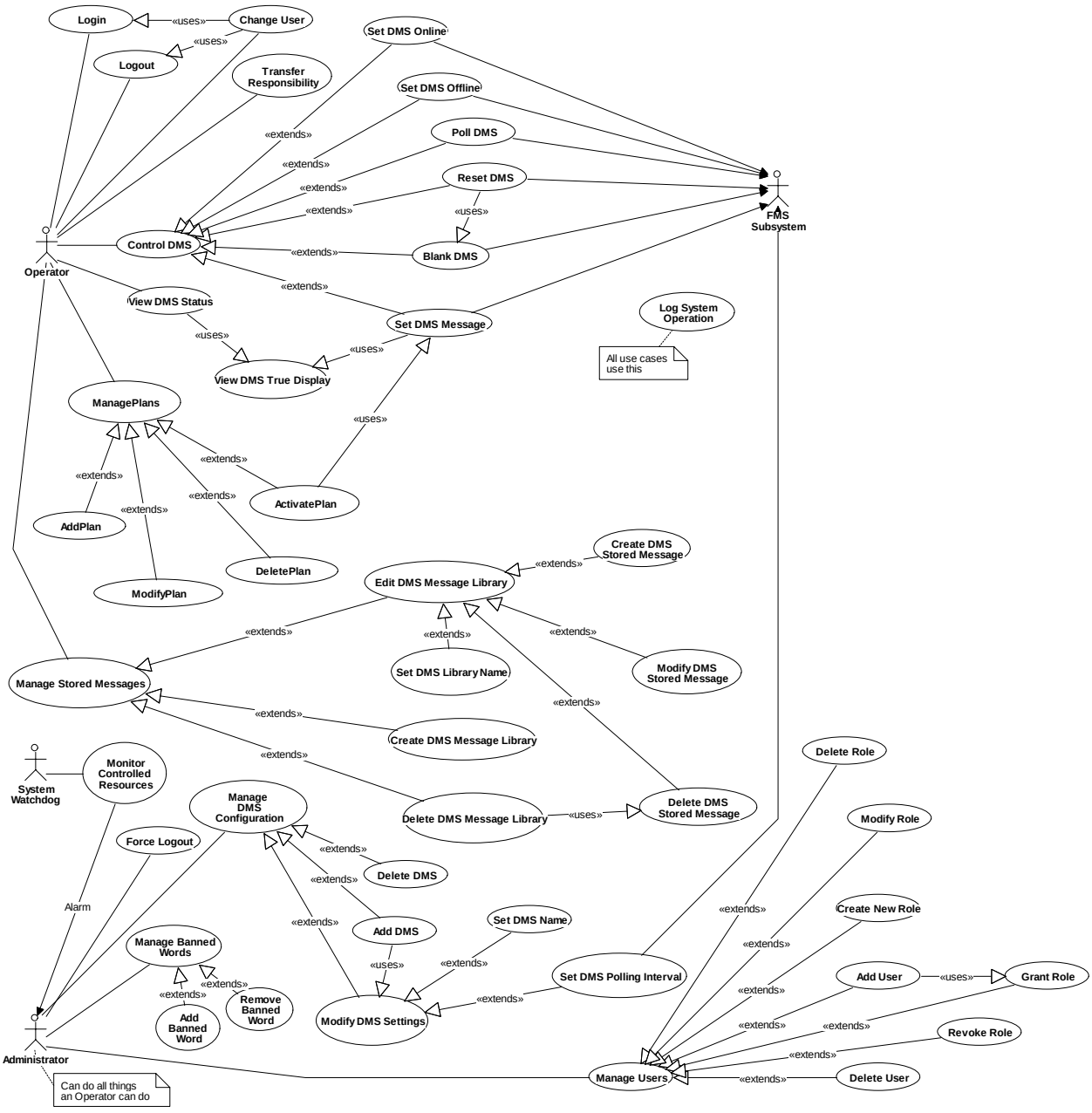

Figure 1. Release1UseCaseDiagram (Use Case Diagram)

4 - Classes

The following class diagram shows the classes required to provide the use cases of the system. The relationships between the classes are shown as well as methods and attributes. Refer to the class descriptions provided below the diagram for an overview of the purpose of each class.

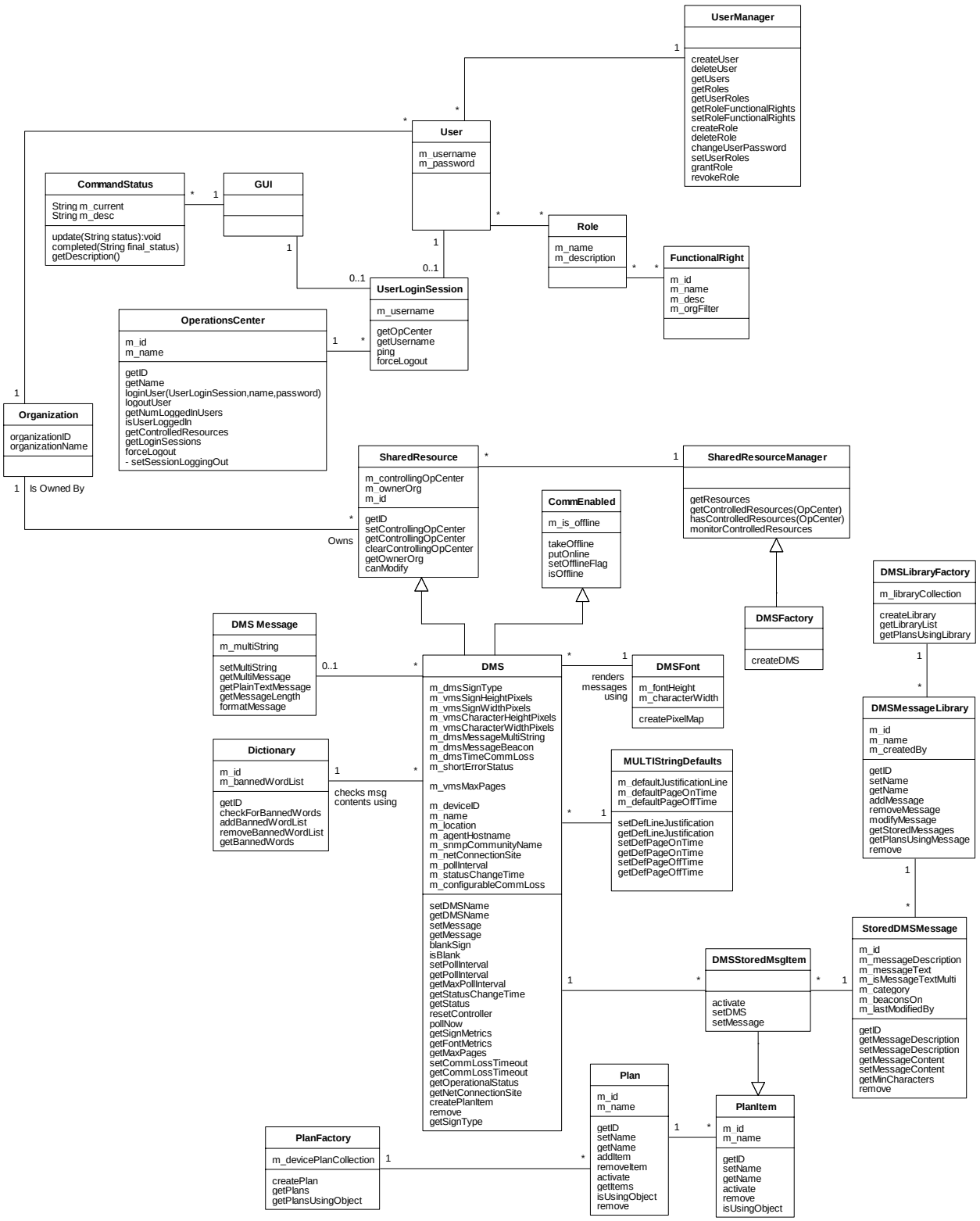

Figure 2. Release1ClassDiagram (Class Diagram)

4.1 CommEnabled (Class)

The CommEnabled interface is implemented by objects that can have their communications turned on or off. This typically only applies to field devices.

4.2 CommandStatus (Class)

The CommandStatus class is used to allow a calling process to be notified of the progress of an asynchronous operation. This is typically used by a GUI when field communications are involved to complete a method call, allowing the GUI to show the user the progress of the operation. The long running operation calls back to the CommandStatus object periodically as the command is executed and makes a final call to the CommandStatus when the operation has completed. The final call to the CommandStatus from the long running operation indicates the success or failure of the command.

4.3 Dictionary (Class)

This class is used to check for banned words in a message that may be displayed on a DMS. In addition to methods for checking the words, it has methods to allow the contents of the dictionary to be changed.

4.4 DMS (Class)

This class represents a Dynamic Message Sign (DMS). It has attributes and methods for controlling and maintaining the status of the DMS within the system.

4.5 DMSFactory (Class)

The DMSFactory provides a means to create new DMS objects to be added to the system.

4.6 DMSFont (Class)

This class contains the functionality for translating text messages into pixels for display on a DMS.

4.7 DMSLibraryFactory (Class)

This class is used to create new DMS libraries and maintain them in a collection.

4.8 DMS Message (Class)

This class represents a text message which is capable of being displayed on a DMS. It contains methods for input and output of the message in different formats.

4.9 DMSMessageLibrary (Class)

This class represents a logical collection of DMS messages which are stored in the database.

4.10 DMSStoredMsgItem (Class)

This class represents a plan item that is used to associate a stored DMS message with a specific DMS. When the item is activated, it sets the message of the DMS to the stored message to which it is linked.

4.11 FunctionalRight (Class)

The FunctionalRights class represents the right to perform an action or set of actions. The functional right can be limited to apply to a single organization's shared resources. If the filter is not used, the functional right applies to all organization's shared resources.

4.12 GUI (Class)

This class represents the Graphical User Interface application at a high level. The GUI uses the CommandStatus class to track the progress of long running operations. The GUI serves a UserLoginSession object to represent a user that is logging in or logged into the system. The GUI provides this UserLoginSession object to the server, allowing the server to monitor the presence of the GUI application.

4.13 MULTIStringDefaults (Class)

This class contains the model-specific default values for creating MULTI strings for a DMS. MULTI is a standard mark-up language specified by NTCIP for specifying how a text message is to be displayed by a DMS.

4.14 OperationsCenter (Class)

The OperationsCenter represents a center where one or more users are located. This class is used to log users into the system. If the username and password provided to the loginUser method are valid, the caller is given a token that contains information about the user and the functional rights of the user. This token is then used to call privileged methods within the system. Shared resources in the system are either available or under the control of an OperationsCenter. The OperationsCenter keeps track of users that are logged in so that it can ensure that the last user does not log out while there are shared resources under its control. This list of logged in users is also available for monitoring system usage or to force users to logout for system maintenance.

4.15 Organization (Class)

The Organization class represents an organization that participates in the Chart system

through ownership of shared resources. The Organization can be used in conjunction with functional rights to determine the level of access users have to shared resources owned by a given organization. This allows access to be granted to a user to perform controlled operations on shared resources owned by one organization but not another.

4.16 Plan (Class)

This class has a collection of Plan Items which it maintains. It has functionality for changing the plan items, and also allows the plan to be activated, which has the effect of activating each plan item in the plan.

4.17 PlanFactory (Class)

This class creates, destroys, and maintains the collection of plans which can be used in the system.

4.18 PlanItem (Class)

This class represents an action within the system that can be planned in advance. This abstract class is subclassed for specific actions that can be planned in the system.

4.19 Role (Class)

A Role is a collection of functional rights. A Role can be granted to a user, thus granting the user all functional rights contained within the role.

4.20 SharedResource (Class)

The SharedResource interface is implemented by any object that must always have an operations center responsible for the disposition of the resource while the resource is in use.

4.21 SharedResourceManager (Class)

The SharedResourceManager interface is implemented by classes that manage shared resources. Implementing classes must be able to provide a list of all shared resources under their management. Implementing classes must also be able to tell others if there are any resources under its management that are controlled by a given operations center.

4.22 StoredDMSMessage (Class)

This class represents a stored DMS message which is created by the DMS Message Editor and stored in the database. It can be displayed on multiple DMS models and contains an attribute stating the minimum width of a sign that can display the message in its entirety.

4.23 User (Class)

The User class represents a Chart II system user. In order to log into the Chart II system, a user must be defined in the user database.

4.24 UserLoginSession (Class)

The UserLoginSession class is used to store information about a user that is logged into the system. This object is served from the GUI and provides a means for the servers to call back into the GUI process.

4.25 UserManager (Class)

The UserManager provides access to data dealing with user management. This includes users, roles, and functional rights. The UserManager is largely an interface to the User Management database tables.

5 – Sequence Diagrams

This section shows a sequence diagram for each Use Case in the system. These sequence diagrams show at a high level how the classes interact to perform each task.

5.1 ActivatePlan:Basic (Sequence Diagram)

A previously created plan that exists in the system can be activated by a user having the proper functional rights. The concept of plans is generic in nature. This diagram shows the activation of a plan containing plan items used to set messages on DMSs. Since activating a plan may involve many field communications, a command status object is used to track the progress of the plan activation. Each plan item in the plan is activated, with a command status object being used to track the progress of each plan item activation. The Plan monitors the progress of each item activation and reports a status back to the caller.

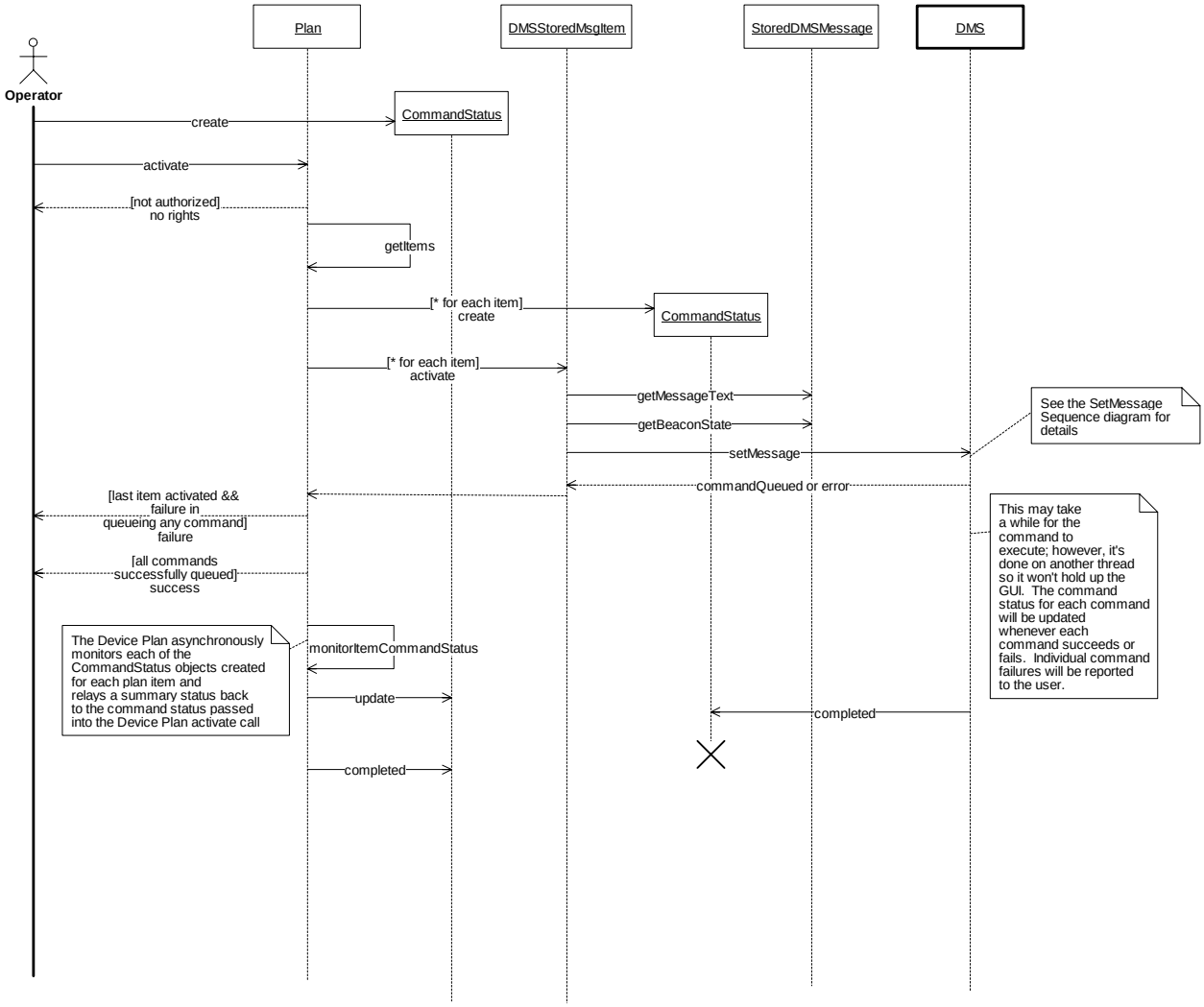

Figure 3. ActivatePlan:Basic (Sequence Diagram)

5.2 AddBannedWord:Basic (Sequence Diagram)

Banned words may be added to the dictionary by a user having the proper functional rights. When words are added to the dictionary, the event service is used to push the changes to the dictionary to interested parties, which would include any process that chooses to cache a local copy of the dictionary to limit the number of network calls for checking message contents. Caching would most likely be used by the GUI so that it can provide dynamic word checking without having to make many network calls while the user is typing. Note that even if a cache of the dictionary is used by a client, all final checking for banned words is done on the server side prior to sending the message to the actual sign.

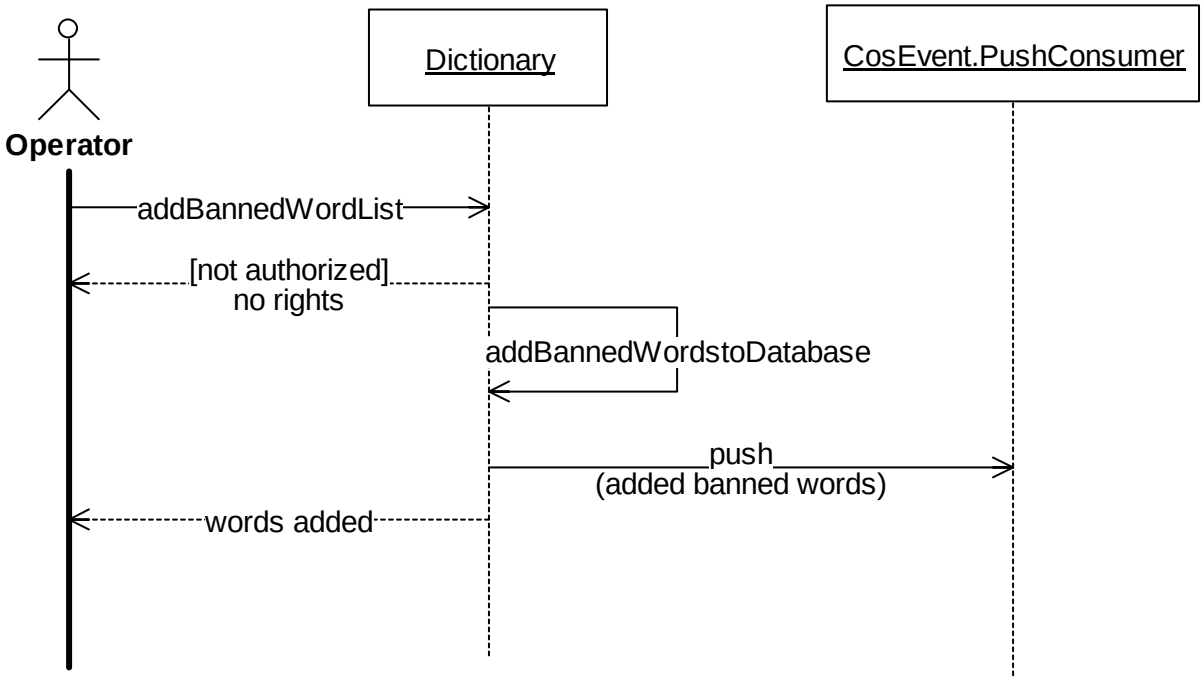

Figure 4. AddBannedWord:Basic (Sequence Diagram)

5.3 AddDMS:Basic (Sequence Diagram)

A DMS is added to the system using the DMS Factory. This involves creating a new DMS object, setting up the proper configuration within the FMS subsystem, publishing the existence of the DMS in the CORBA trading service, and notifying interested parties of the new DMS via the CORBA event service.

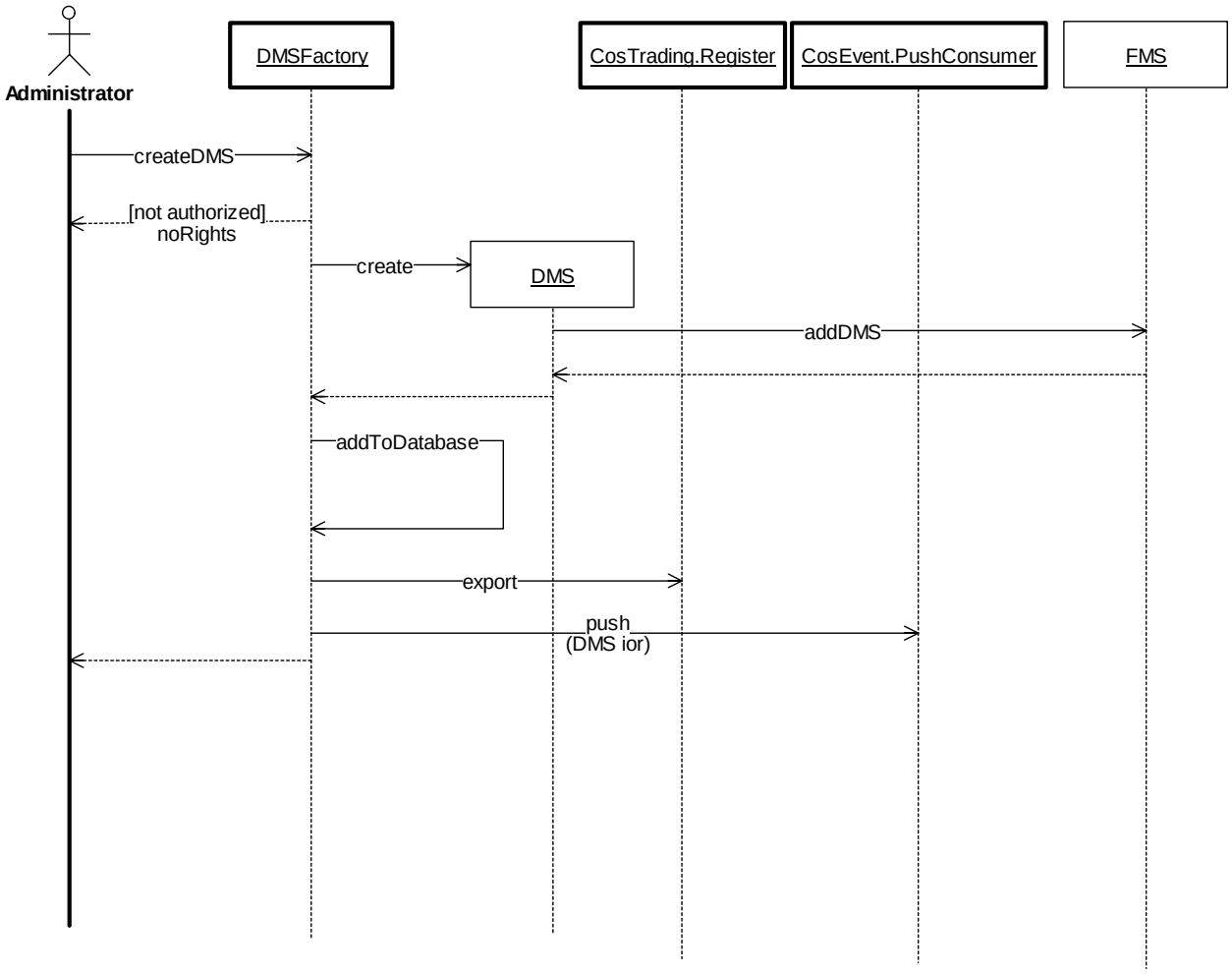

Figure 5. AddDMS:Basic (Sequence Diagram)

5.4 AddPlan:Basic (Sequence Diagram)

A user with the proper functional rights may add a plan to the system. Once the plan is added to the system, plan items may be added to it using the ModifyPlan use case.

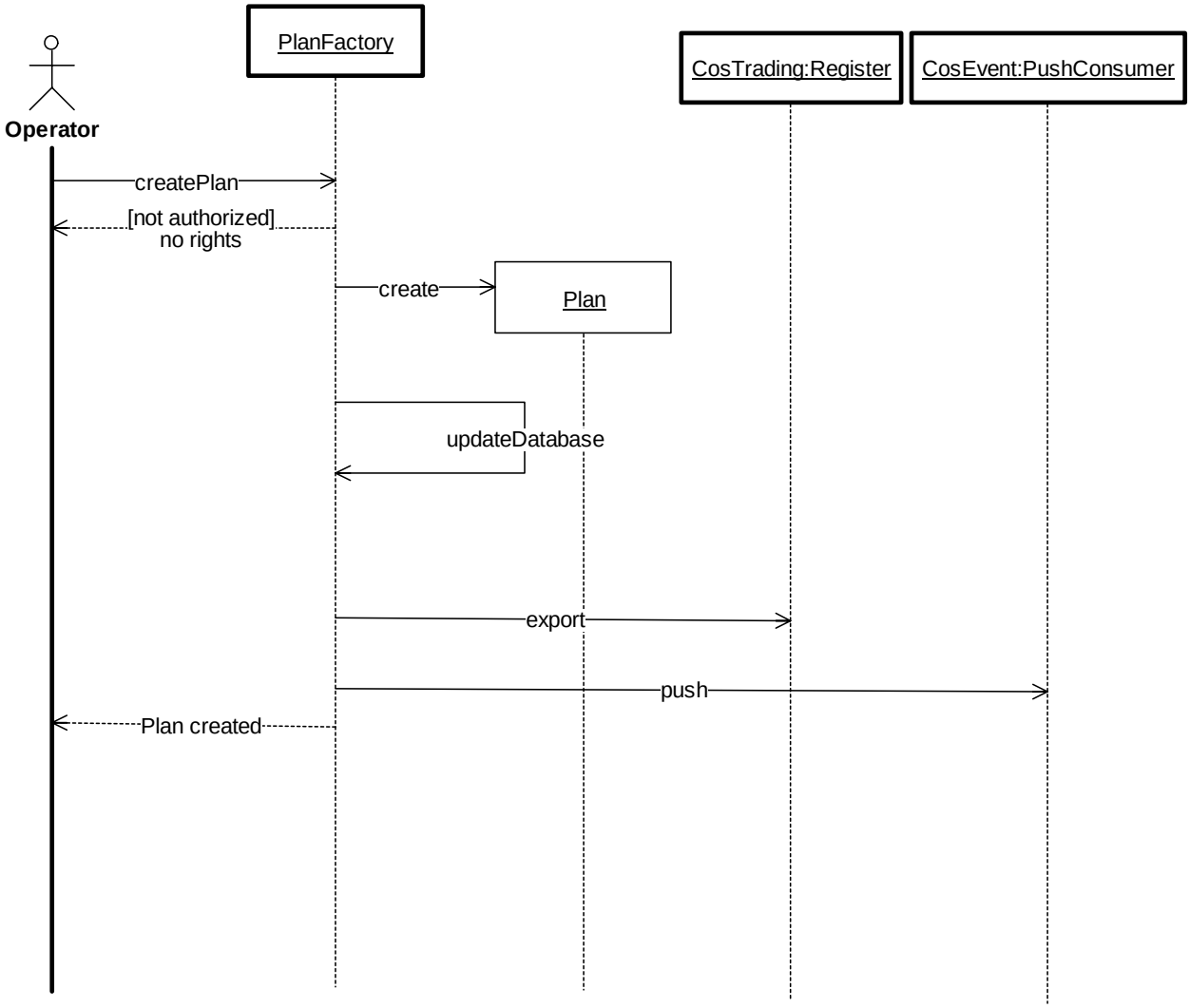

Figure 6. AddPlan:Basic (Sequence Diagram)

5.5 AddUser:AddUser (Sequence Diagram)

A user possesing the proper functional rights may add another user to the system. This involves storing the user configuration information in the database.

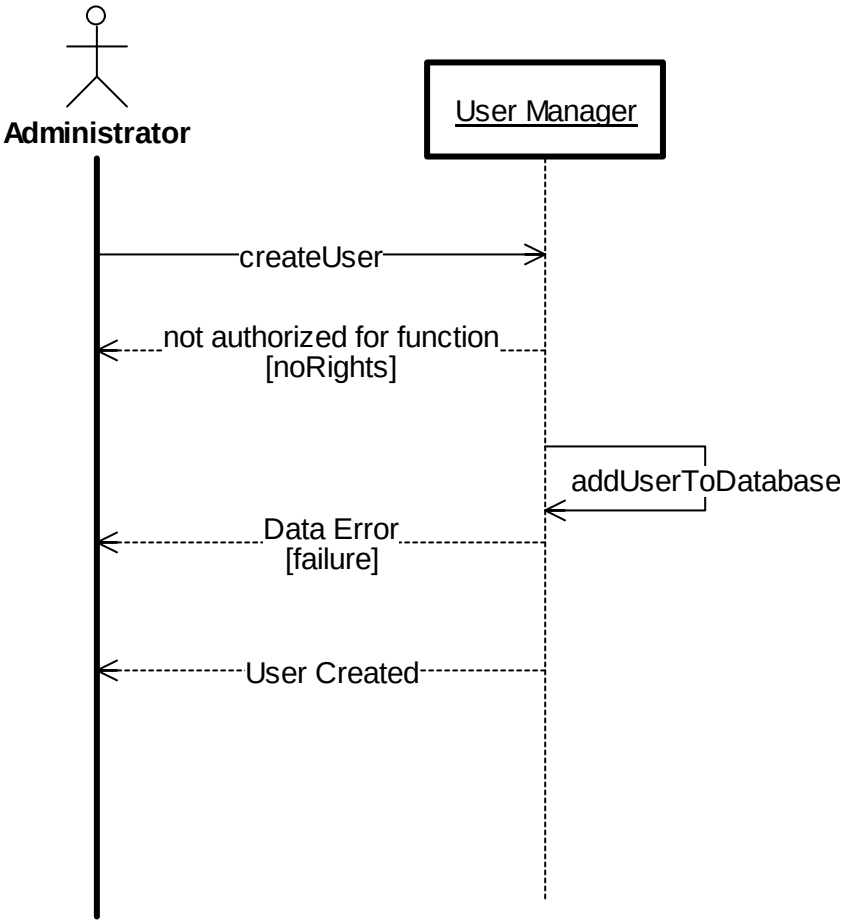

Figure 7. AddUser:AddUser (Sequence Diagram)

5.6 BlankDMS:Basic (Sequence Diagram)

A user possessing the proper functional rights may blank the display (including beacons) of a DMS. DMS objects that are offline cannot be communicated with. Blanking a DMS also involves the concept of shared resource management, allowing only the controlling operations center to blank a DMS (unless the user has the override functional right). Once clear to perform the operation, the command is queued within the DMS object and the user is notified that a long running operation is in progress. The supplied CommandStatus object is used to notify the caller of the ongoing progress. When the DMS pulls the blank command off its internal queue, it executes the command using the FMS subsystem. If the command is successful, the controlling operations center is cleared since the DMS is no longer being used to display a message. The CORBA event service is used to push state changes of the DMS, for both the action of the sign being blanked and the controlling operations center being removed.

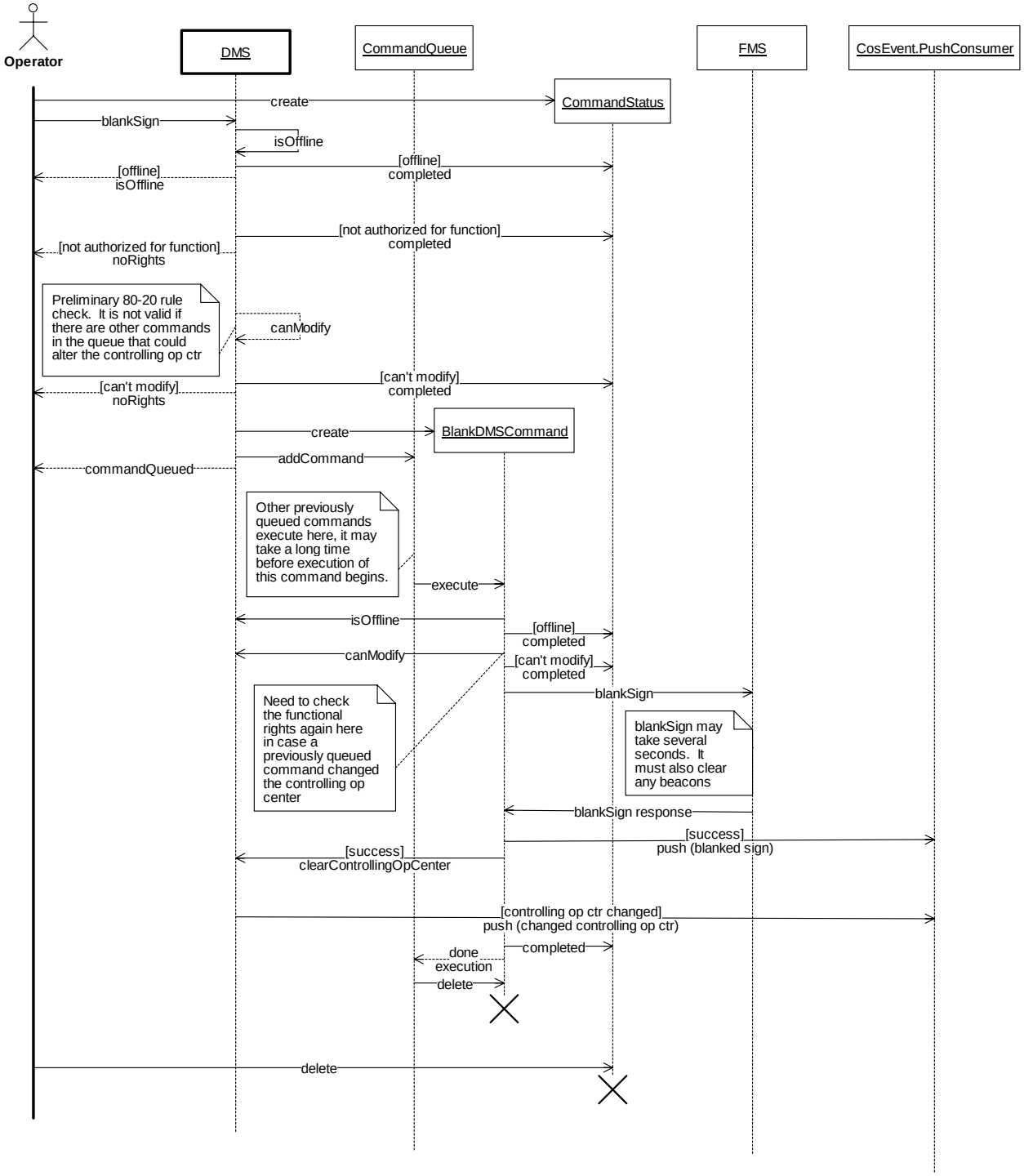

Figure 8. BlankDMS:Basic (Sequence Diagram)

5.7 ChangeUser:Basic (Sequence Diagram)

There is a requirement for someone to be logged into the system at an operations center at all times that the operations center is in control of any shared resources in the system. To accomodate shift changes when there is only one user logged in at the operations center, a feature is provided to allow the user to be changed without actually logging out. Internally, this is accomplished by actually having two users logged in during the transition, logging in the second user prior to logging out the first.

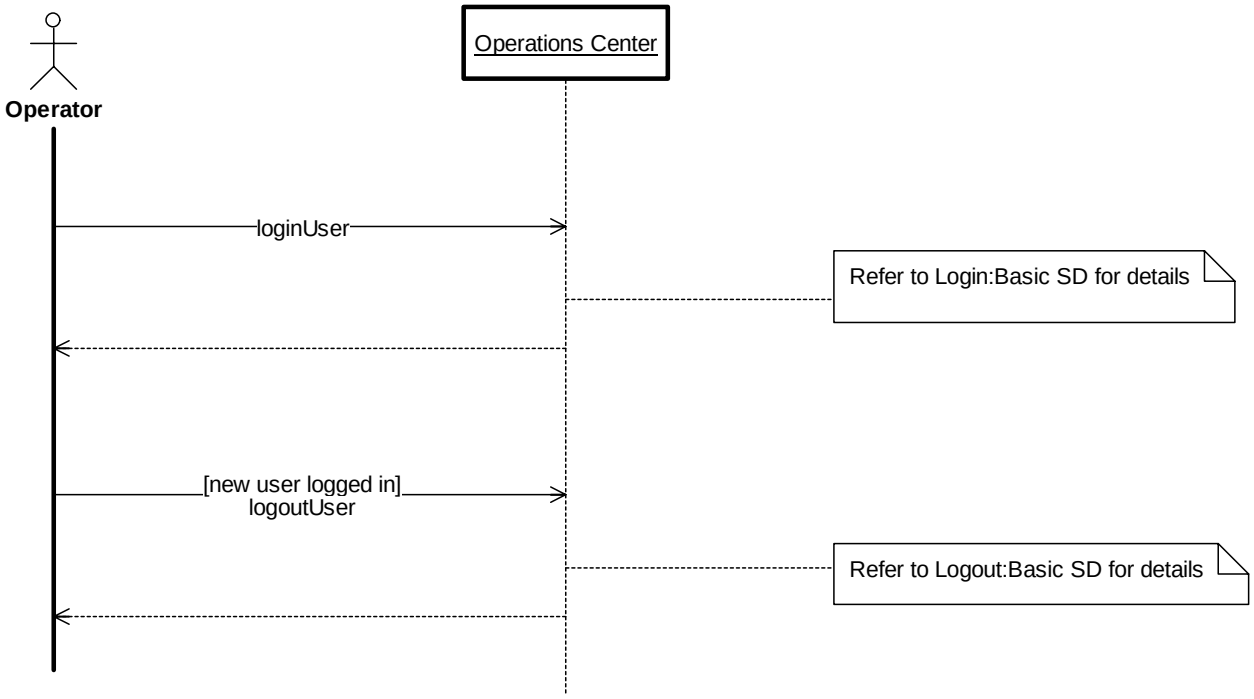

Figure 9. ChangeUser:Basic (Sequence Diagram)

5.8 CreateDMSMessageLibrary:Basic (Sequence Diagram)

A user possessing the proper functional rights can add a DMS Message Library to the system. The library object is created and published via the CORBA Trading Service. An event is pushed via the CORBA Event Service to notify interested parties of the new library.

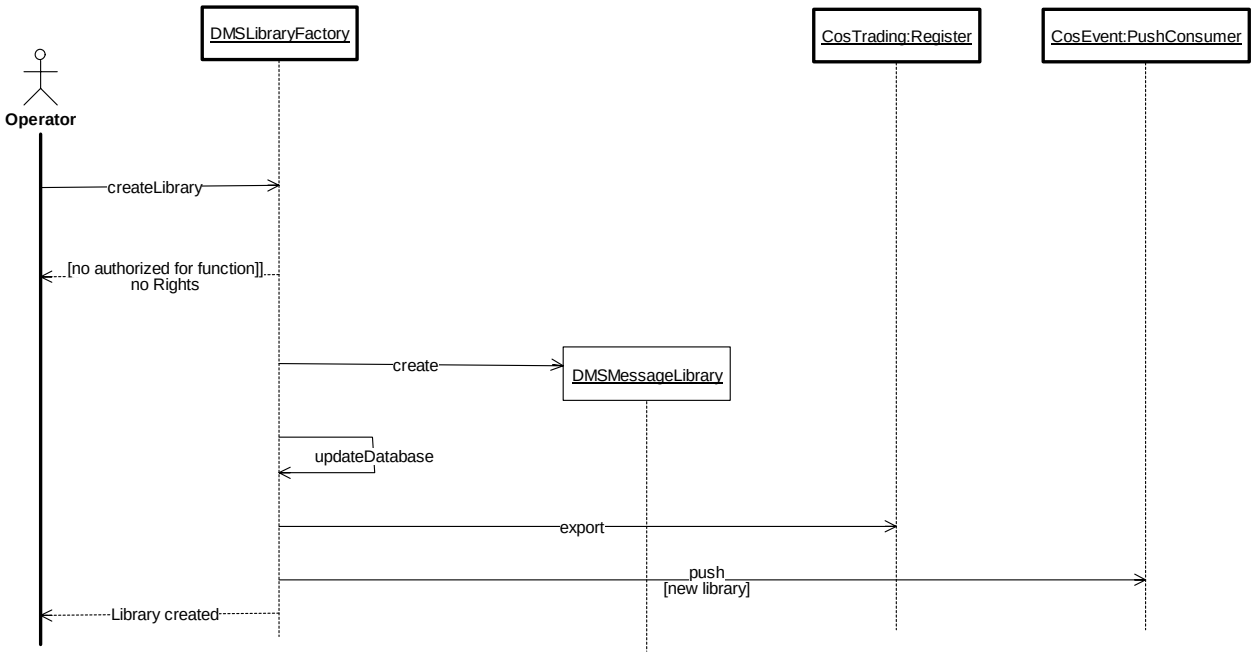

Figure 10. CreateDMSMessageLibrary:Basic (Sequence Diagram)

5.9 CreateDMSStoredMessage:Basic (Sequence Diagram)

A user with the proper functional rights can create a new DMS message to be stored for later use. The contents of the message are checked against a dictionary prior to storing. If approved, the messsage is stored and its existence is pushed to interested parties via the CORBA Event Service. Note that even though a dictionary check is done at the time of storage, the dictionary is always checked on the server side prior to allowing a message to be set on a DMS. Note also that stored messages are not published in the trader and are instead accessed through the library in which they are contained.

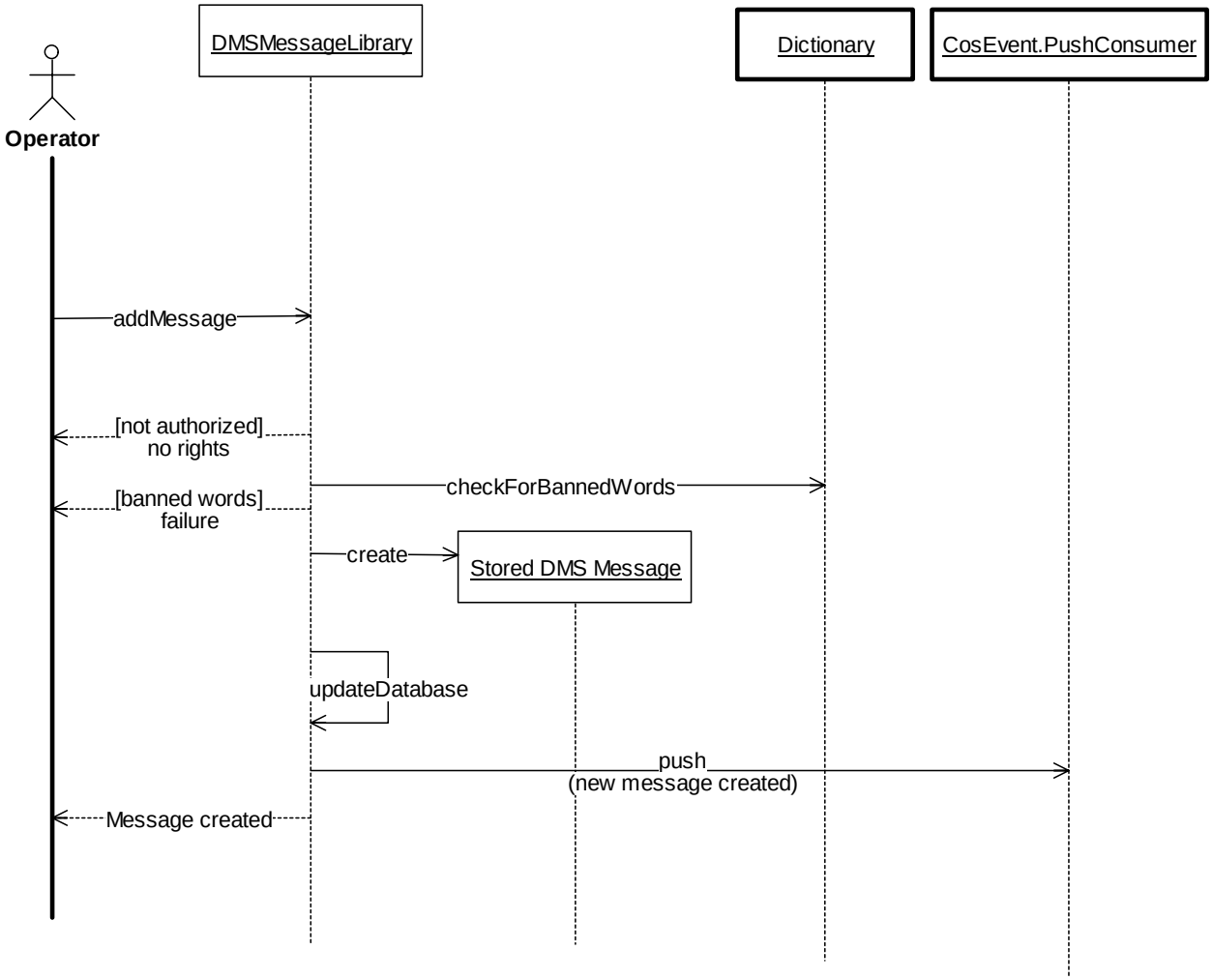

Figure 11. CreateDMSStoredMessage:Basic (Sequence Diagram)

5.10 CreateNewRole:Basic (Sequence Diagram)

A user with the proper functional rights can add a role to the system. This role is used as a collection of functional rights that can easily be assigned to users.

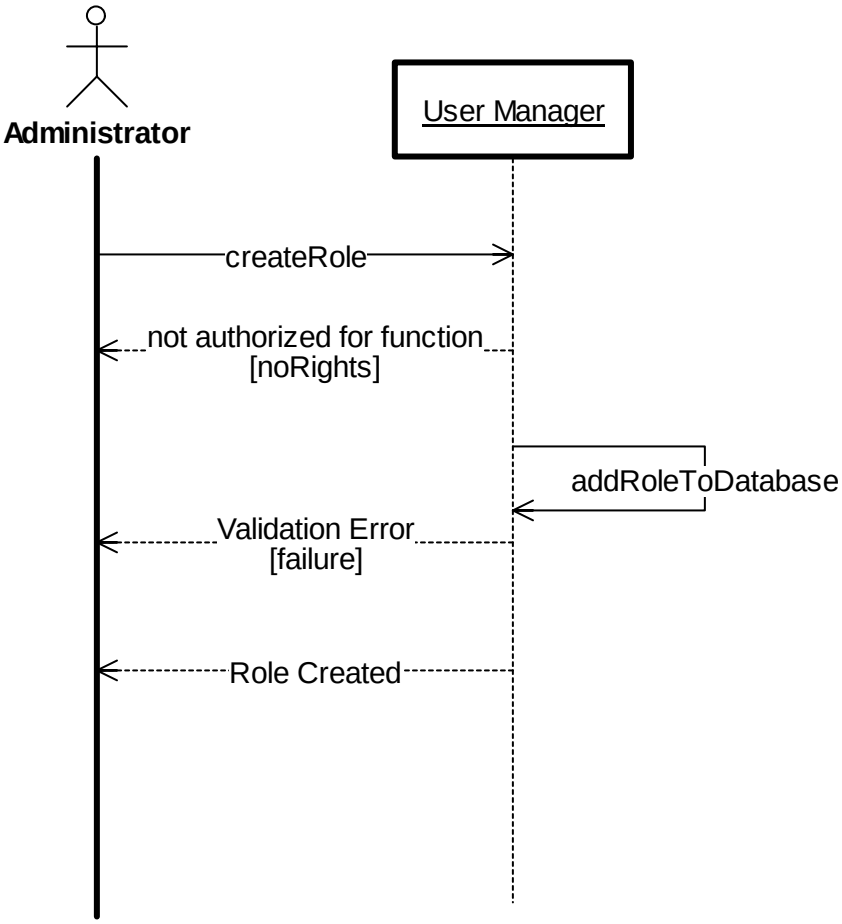

Figure 12. CreateNewRole:Basic (Sequence Diagram)

5.11 DeleteDMS:Basic (Sequence Diagram)

A user with the proper functional rights can remove a DMS from the system. This involves revoking the offer of the DMS from the CORBA Trading Service, removing configuration data from the FMS subsystem, and pushing an event via the CORBA event service.

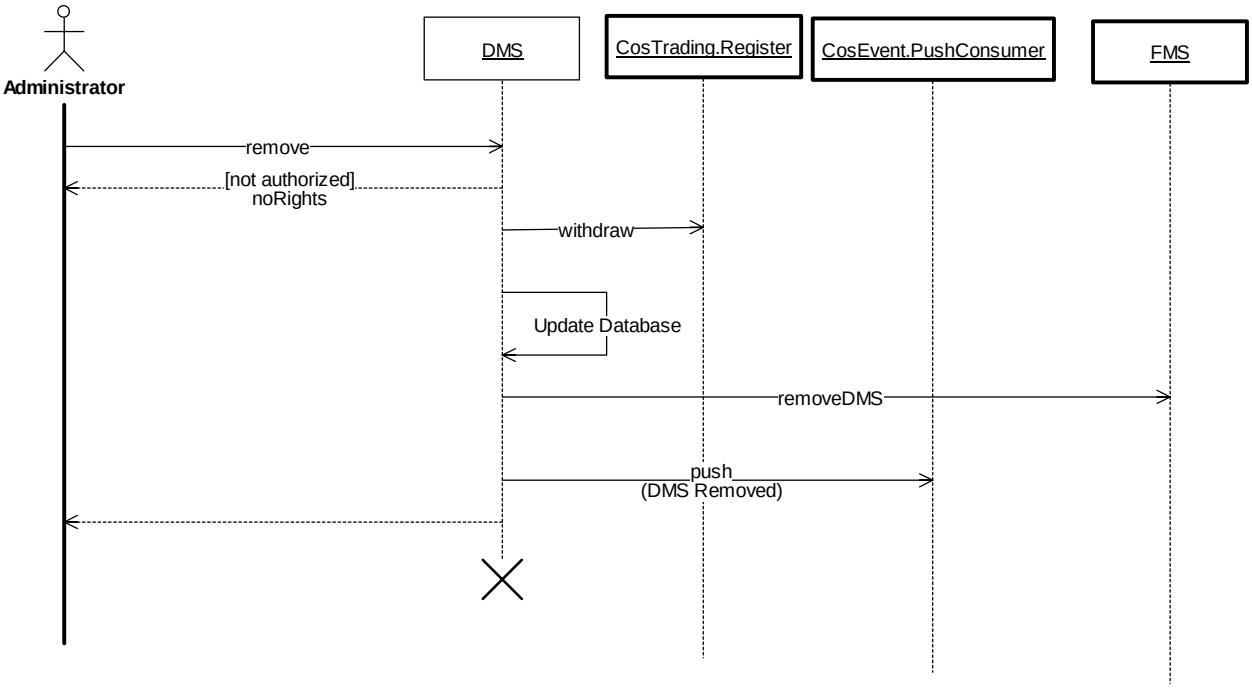

Figure 13. DeleteDMS:Basic (Sequence Diagram)

5.12 DeleteDMSMessageLibrary:Basic (Sequence Diagram)

A user with the proper functional rights can remove a DMS Message Library from the system. This will include the removal of all stored messages contained within the library. Since stored messages may be used in Plans that contain DMSStoredMessageItems, a check is made for any plans that may contain the stored messages being deleted and the user is warned. If the user acknowledges the deletions, each message within the library is removed, events are pushed to notify others the action, and the library is removed from the Trading Service.

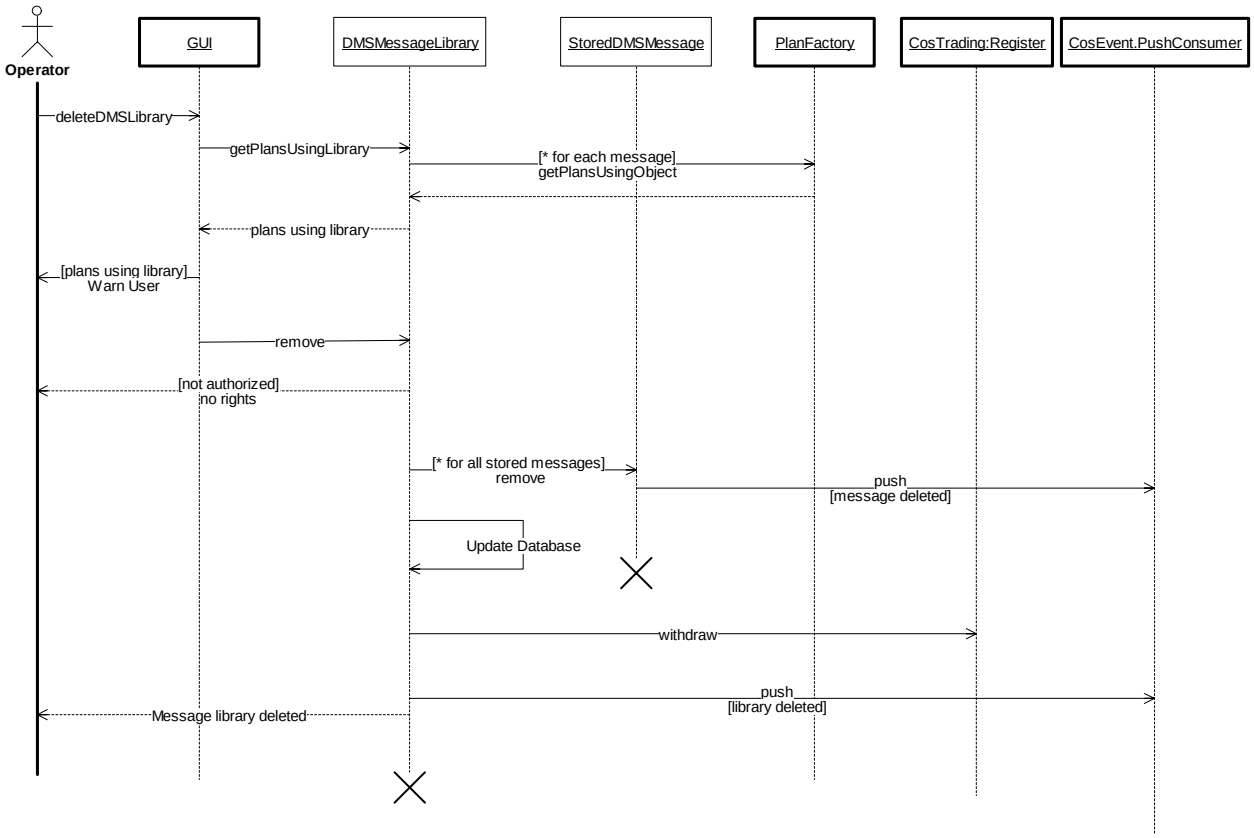

Figure 14. DeleteDMSMessageLibrary:Basic (Sequence Diagram)

5.13 DeleteDMSStoredMessage:Basic (Sequence Diagram)

A user with the proper functional rights may remove a stored DMS message from the system. Since a stored DMS message may be used in a plan, a check is made to see if the message is used in a plan so that the user can be warned accordingly. The act of deleting the stored message involves updating the database and pushing an event to notify others that the message has been removed from its library.

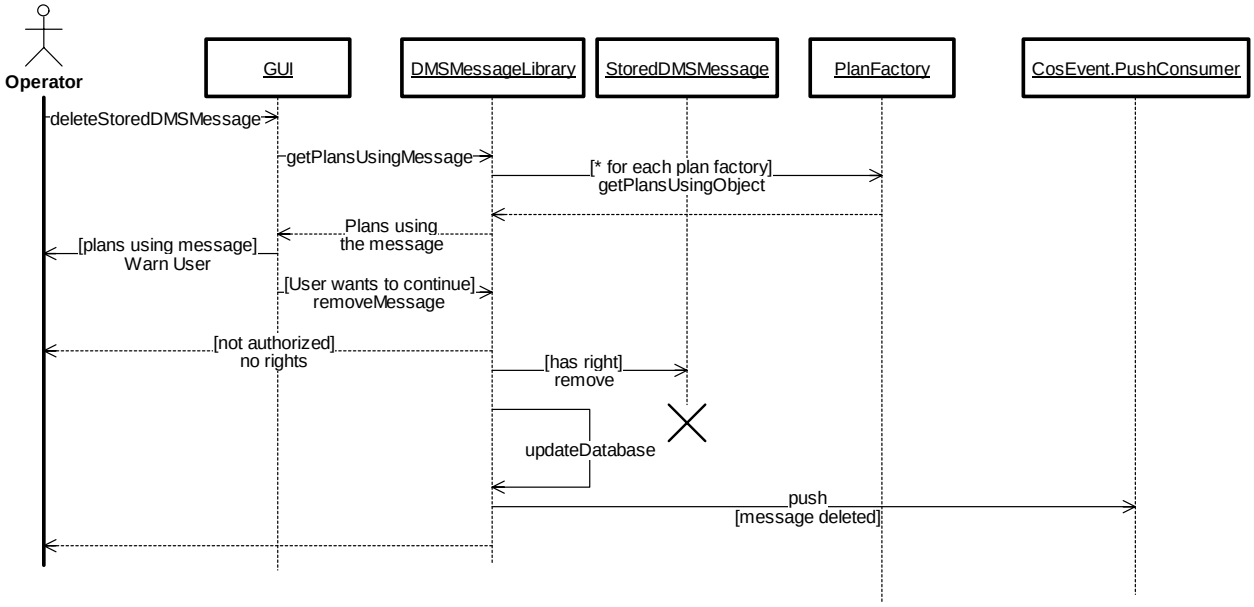

Figure 15. DeleteDMSStoredMessage:Basic (Sequence Diagram)

5.14 DeletePlan:Basic (Sequence Diagram)

A user with the proper functional rights may remove a plan from the system. Each item in the plan is removed, followed by the removal of the plan itself. The Plan is withdrawn from the CORBA trading service, and an event is pushed to notify others of the plan's removal.

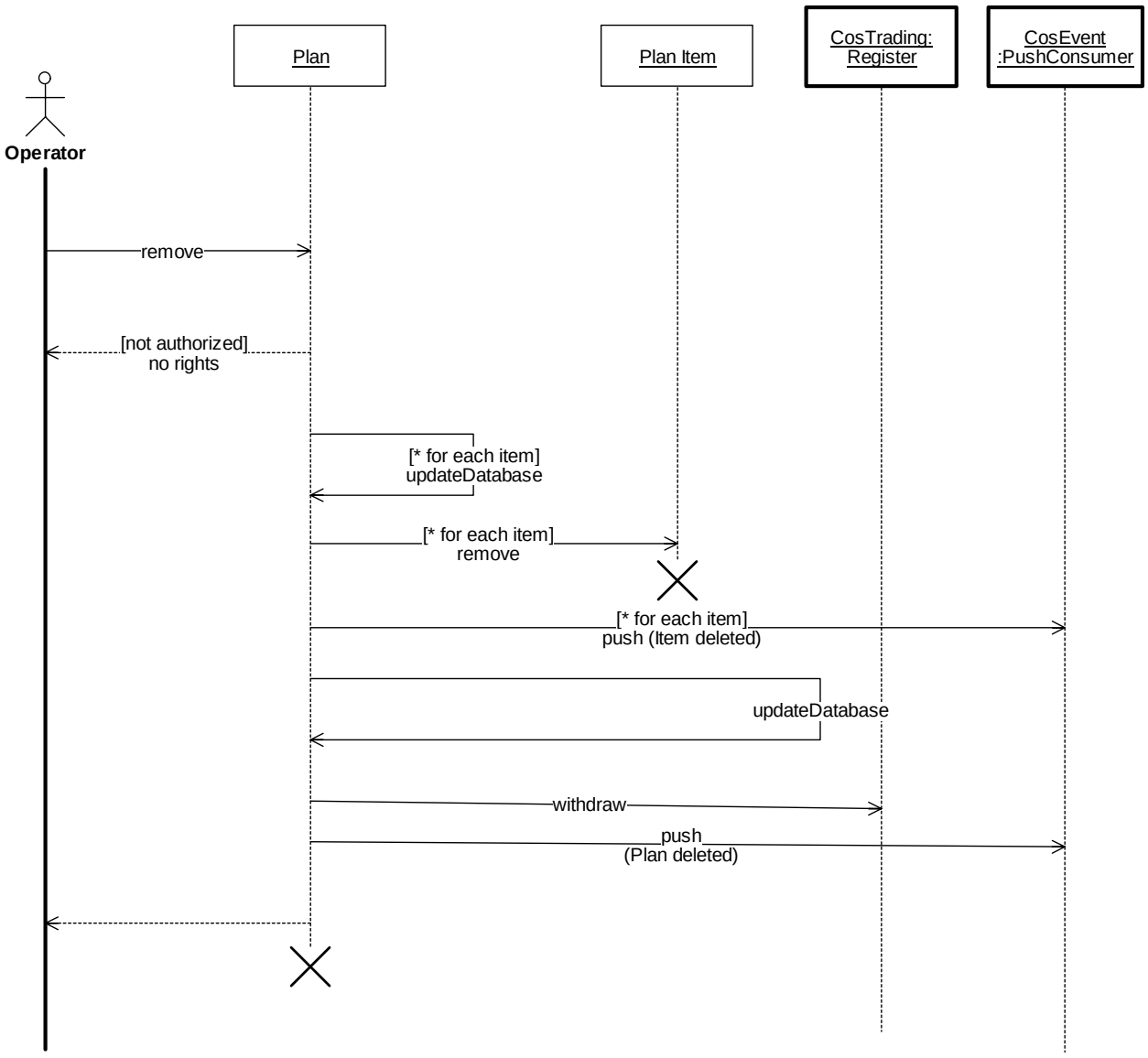

Figure 16. DeletePlan:Basic (Sequence Diagram)

5.15 DeleteRole:Basic (Sequence Diagram)

A user with the proper functional rights may delete a role from the system. The role cannot be deleted unless there are no users currently assigned to the role. Note - roles are not published items, thus no offer exists in the trading service and there is no need to push an event.

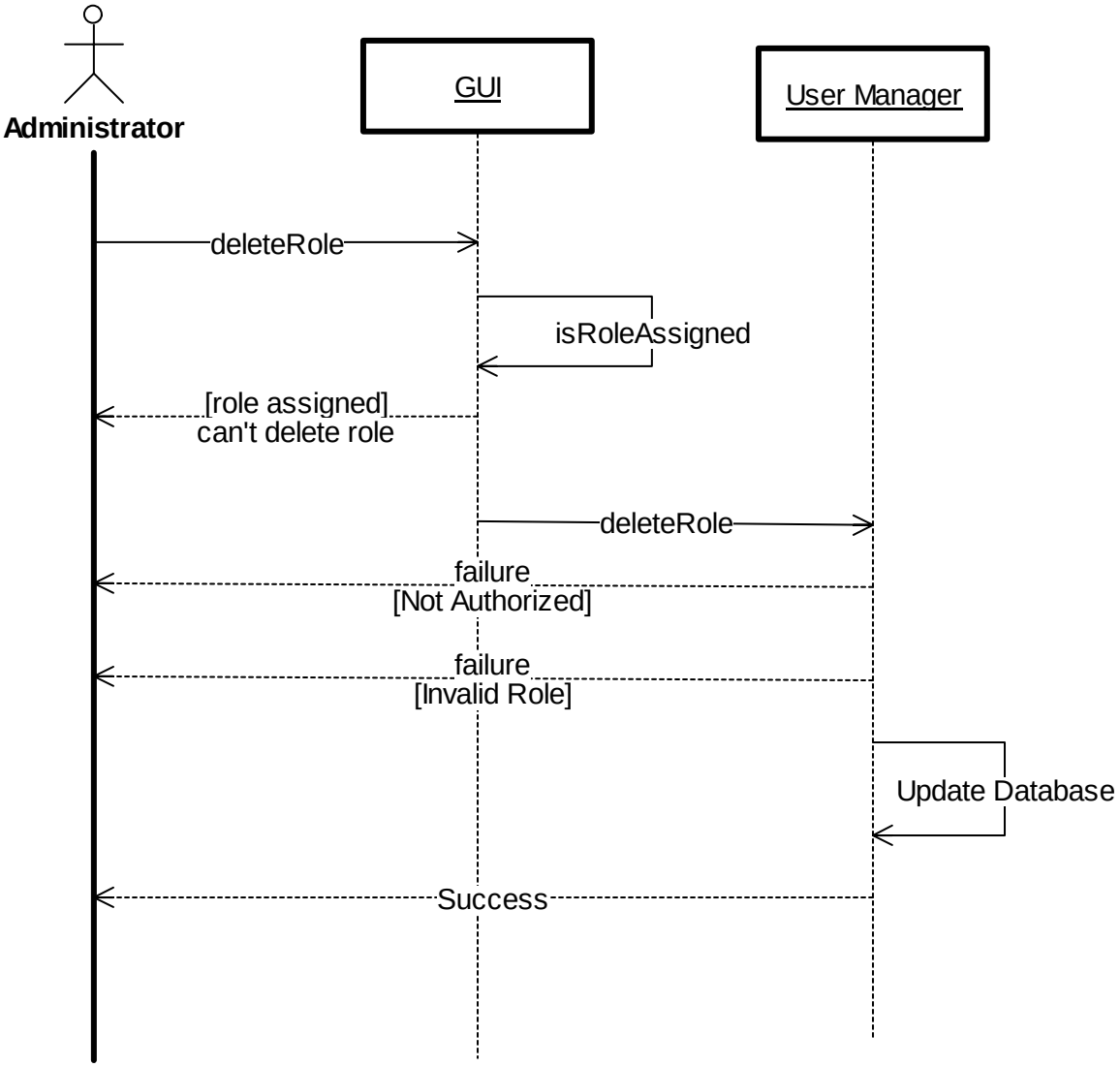

Figure 17. DeleteRole:Basic (Sequence Diagram)

5.16 DeleteUser:Basic (Sequence Diagram)

A user with the proper functional rights may delete a user from the system. Before deleting the user, a check is made to see if the user is logged into any of the operations centers that exist in the system. A user cannot be deleted if currently logged in. If not logged in, the user's information is deleted from the system.

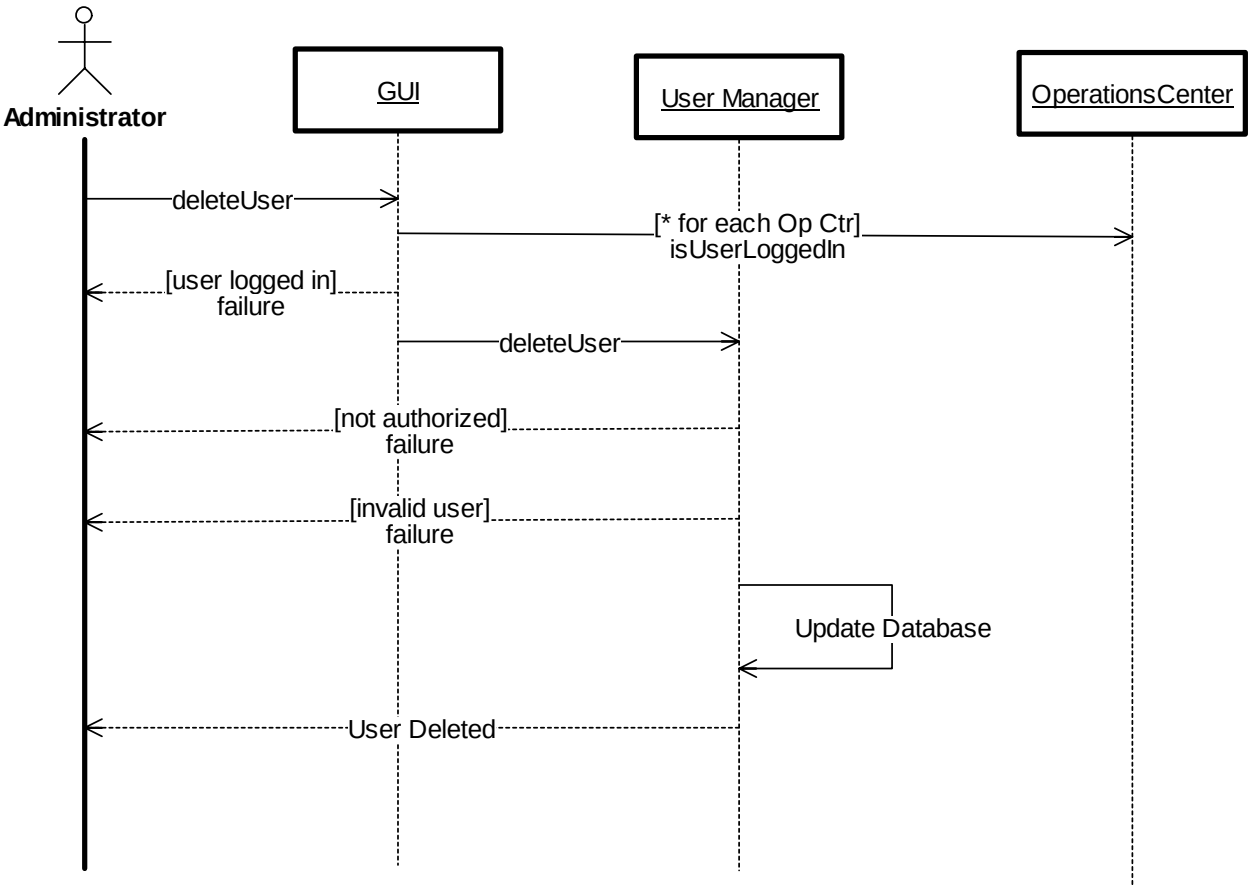

Figure 18. DeleteUser:Basic (Sequence Diagram)

5.17 ForceLogout:Basic (Sequence Diagram)

A user with the proper functional rights may force another user in the system to be logged out. The login session of the user to be logged out is obtained from the user's operation center. The user's login session is used to force the user's GUI to logout.

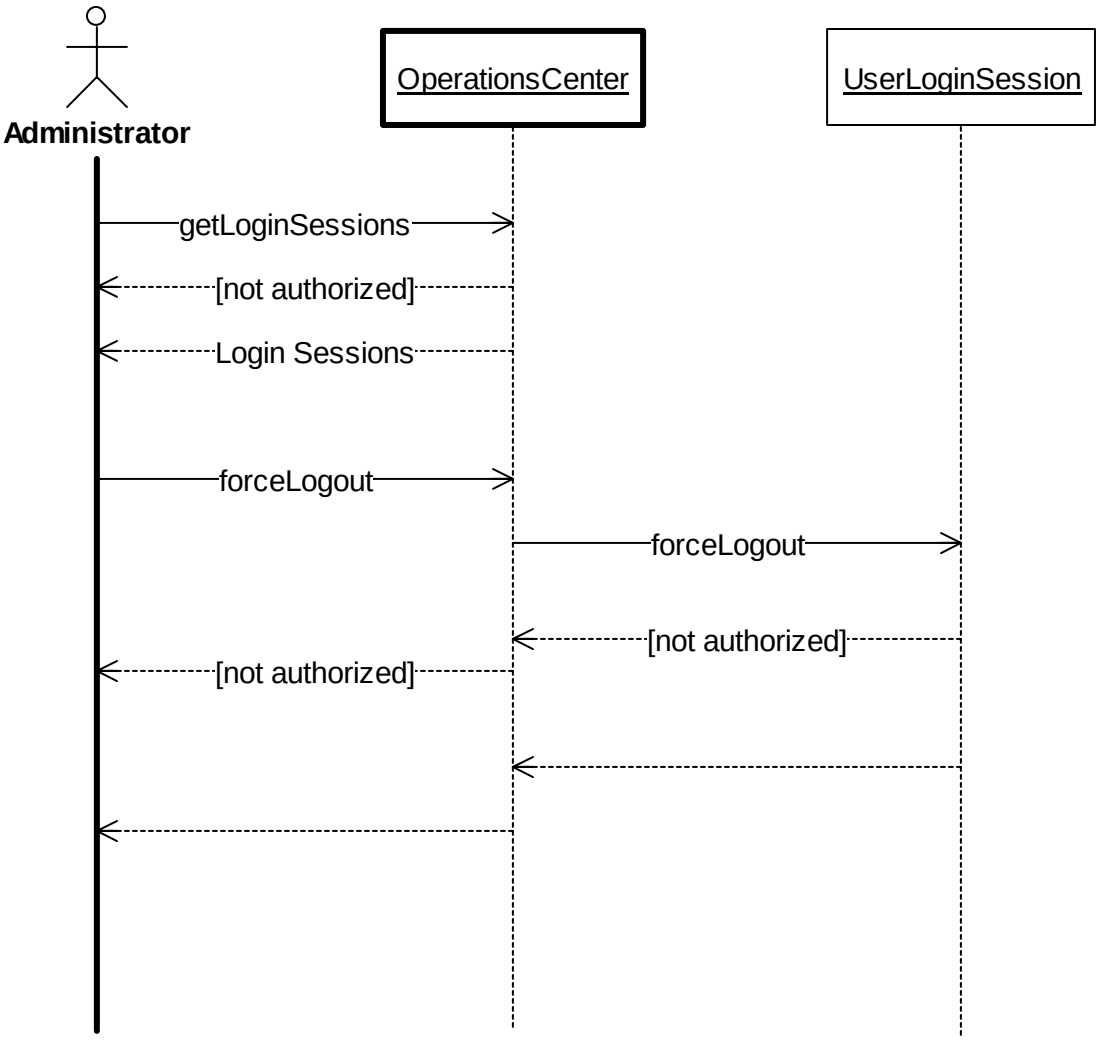

Figure 19. ForceLogout:Basic (Sequence Diagram)

5.18 GrantRole:Basic (Sequence Diagram)

A user with the proper functional rights can grant a role (and thus a collection of functional rights) to a user. The newly granted role is used in conjunction with other roles that are already granted to the user (if any).

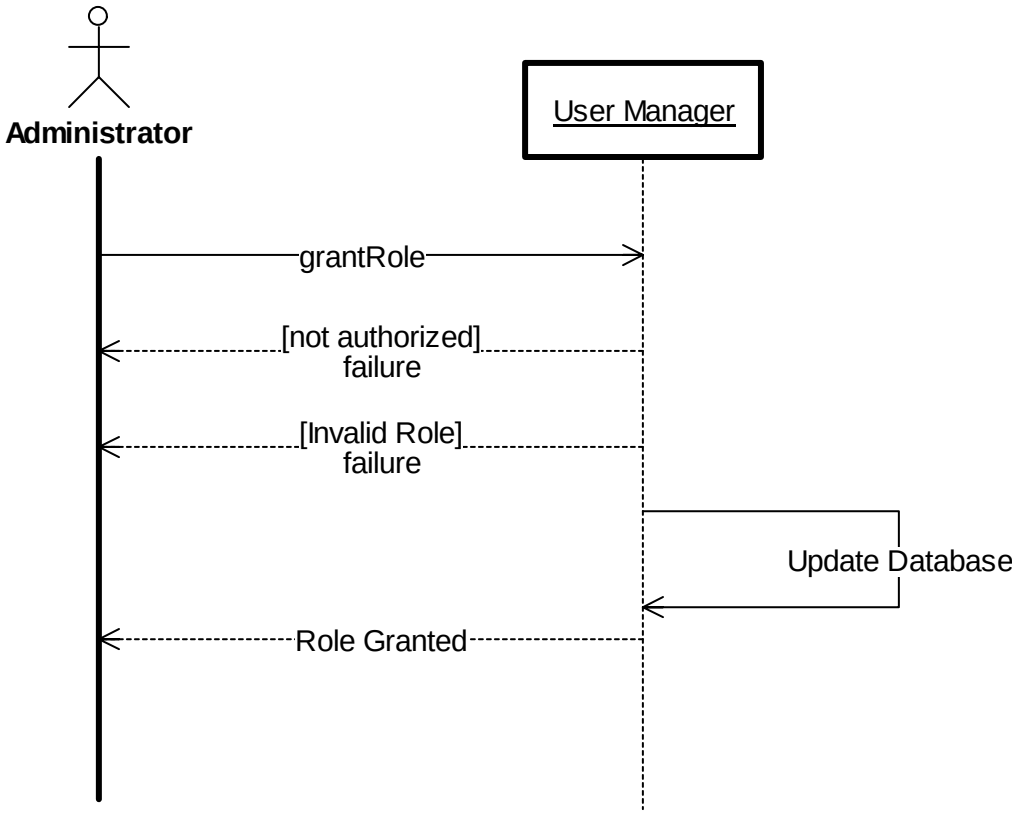

Figure 20. GrantRole:Basic (Sequence Diagram)

5.19 Login:Basic (Sequence Diagram)

Prior to logging into the system, the GUI must create a UserLoginSession object that is used by the OperationsCenter to monitor the application through which the user is using the system. The OperationsCenter loginUser method is used to login to the system. If a valid username and password are given, the OperationsCenter returns a token that is used by the user's GUI to access priveleged operations within the system.

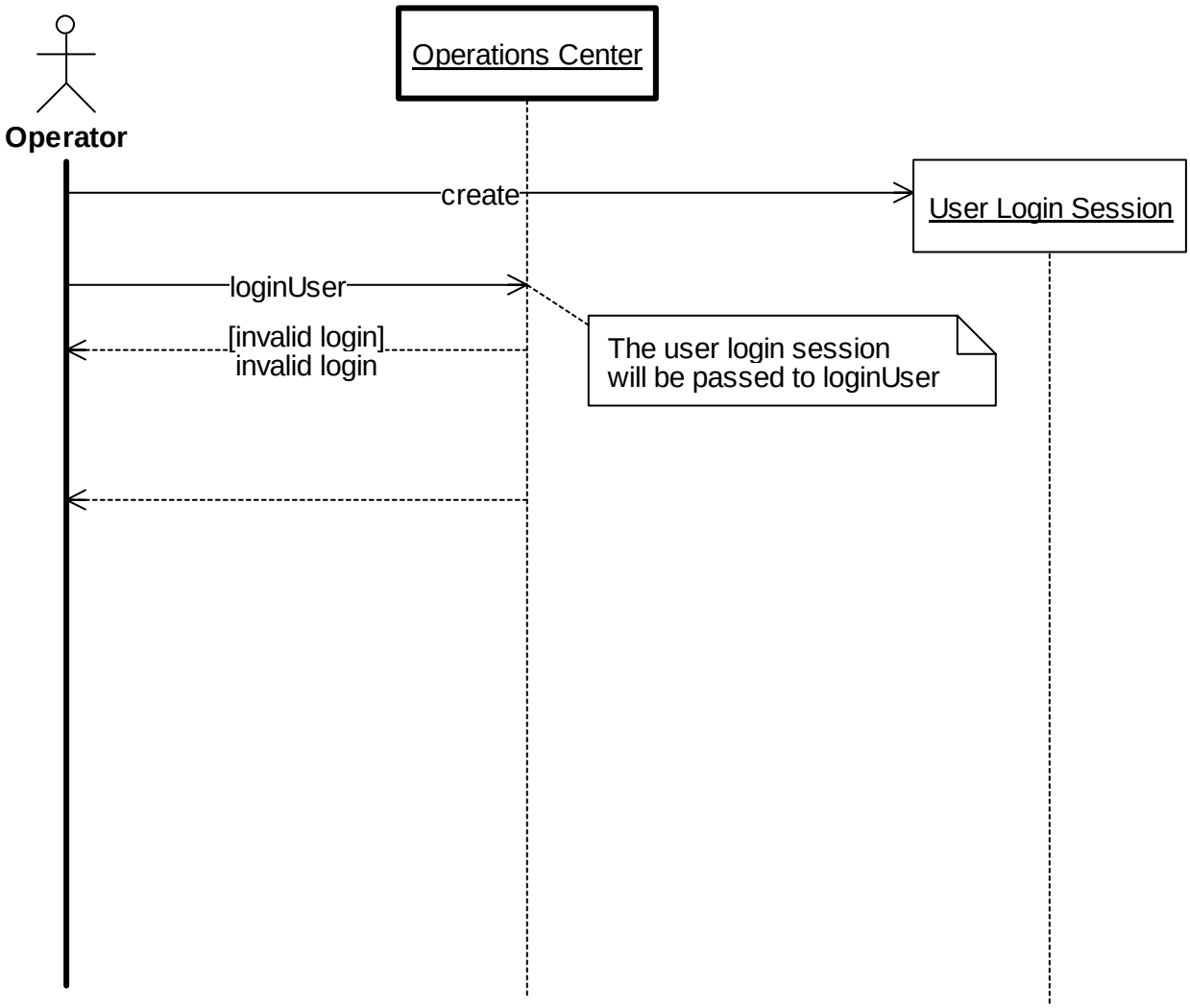

Figure 21. Login:Basic (Sequence Diagram)

5.20 Logout:Basic (Sequence Diagram)

When a user logs out of the system, the operations center where the user was logged in checks to see if the user is the last user remaining at the operations center. If the user is the last user at the operations center, the operations center checks to make sure the operations center is not controlling any shared resources in the system. If it is, the user is not allowed to log out.

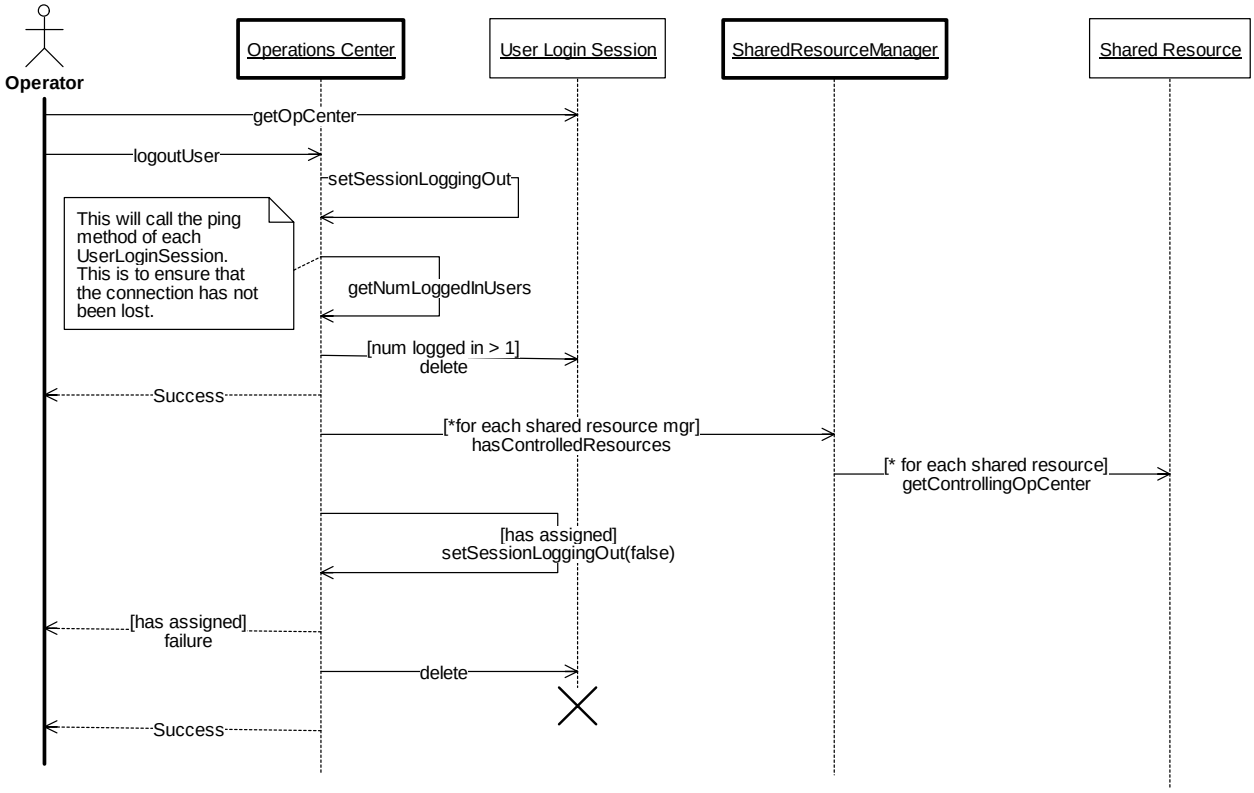

Figure 22. Logout:Basic (Sequence Diagram)

5.21 ModifyDMSStoredMessage:Basic (Sequence Diagram)

A user with the proper functional rights can edit a stored message. The proposed contents for the stored message are checked against the dictionary prior to allowing the new content to be set. The state of the beacons associated with the message are also checked to make sure the beacons are not turned on for a message with no text. An event is pushed via the CORBA Event Service to notify others of the change to the stored message's contents.

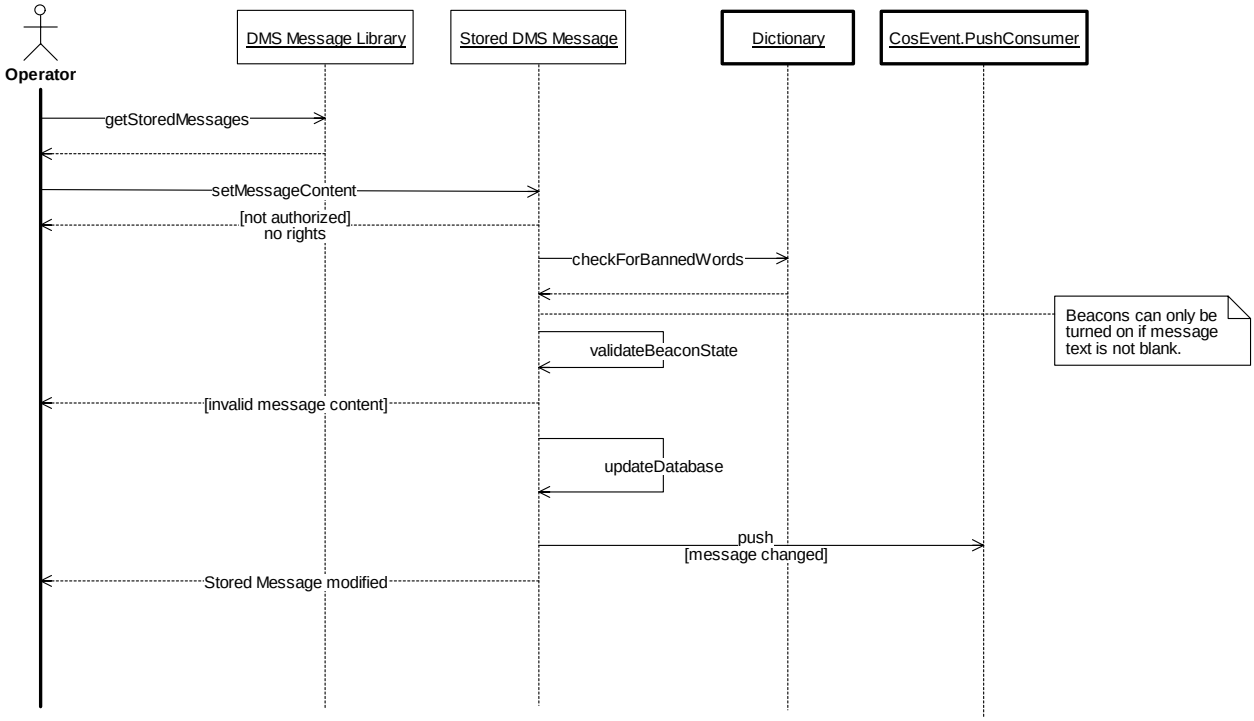

Figure 23. ModifyDMSStoredMessage:Basic (Sequence Diagram)

5.22 ModifyPlan:Basic (Sequence Diagram)

A user with the proper functional rights may modify the contents of a plan. Since the concept of a plan is generic, the sequence diagram shows the modification of a plan that contains DMSStoredMessageItems. The modification process involves obtaining a reference to the item from the plan and invoking methods directly on the plan item. For a DMSStoredMessageItem, this involves setting the stored message or the DMS associated with the stored message. Event notification and database updates are handled by the plan item.

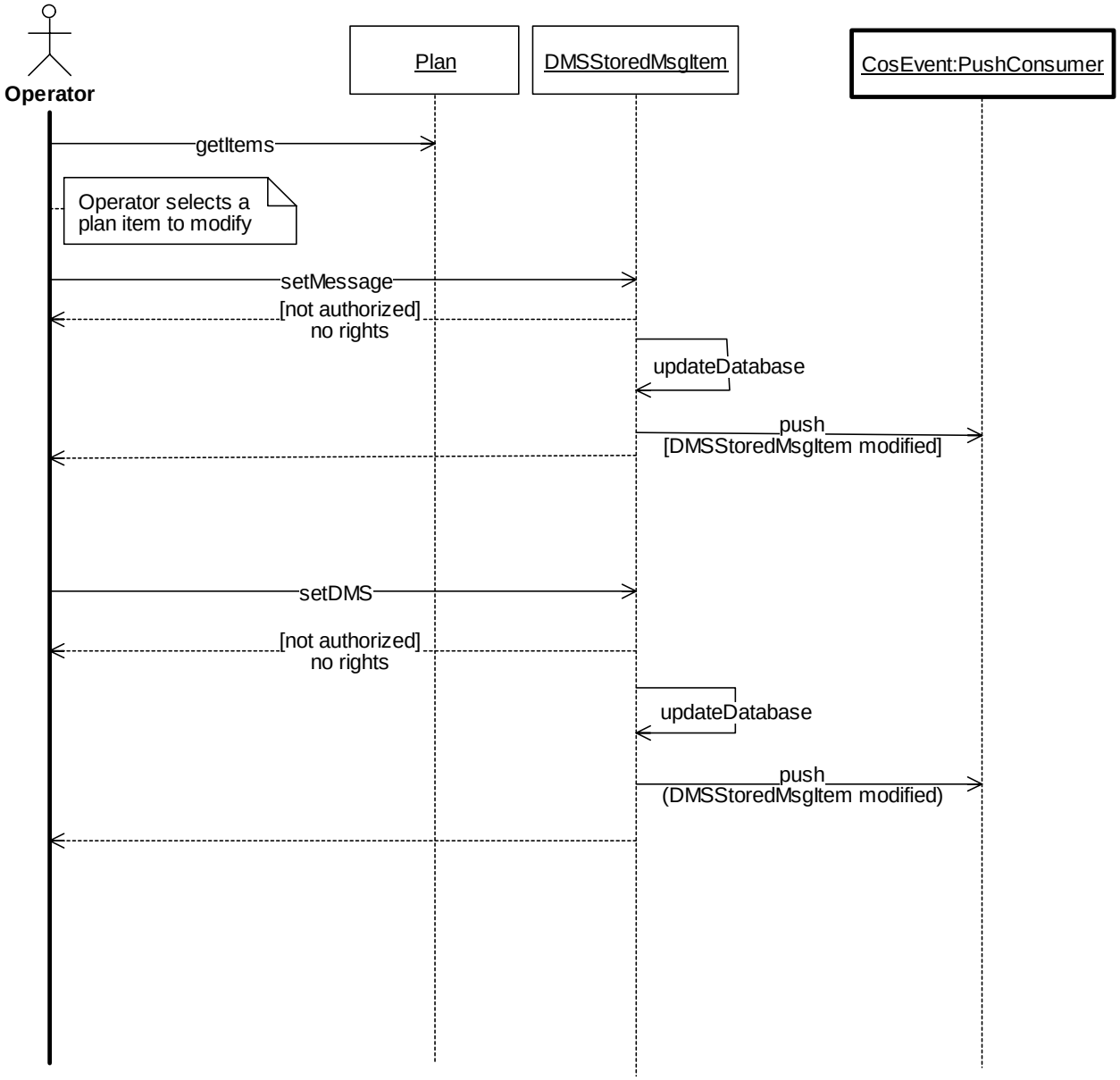

Figure 24. ModifyPlan:Basic (Sequence Diagram)

5.23 ModifyRole:Basic (Sequence Diagram)

A user with the proper functional rights may modify the functional rights assigned to a role.

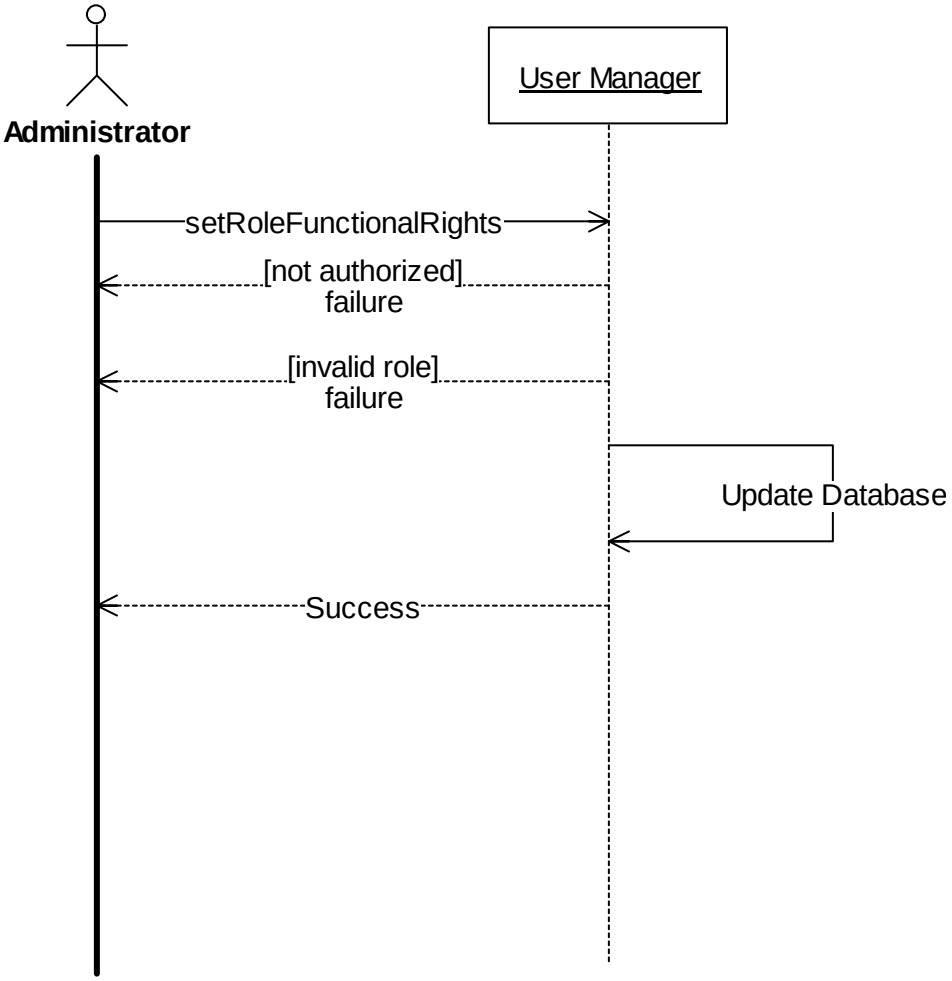

Figure 25. ModifyRole:Basic (Sequence Diagram)

5.24 MonitorControlledResources:Basic (Sequence Diagram)

There is a requirement that all shared resources that are in use have an operations center responsible for them. The system enforces this rule as much as possible, however a monitor is used to detect rare occasions that may occur, such as a power outage at an operations center that is in control of one or more shared resources. The process of monitoring shared resources is delegated to each shared resource manager that exists in the system. Periodically, a shared resource manager checks each of its resources for controlling operations centers. It uses a summary of the operations centers that are controlling its resources and calls each operations center to check the number of logged in users at the operations center. The operations center checks each user login session to make sure it is still alive when it determines its count of logged in users. Should a shared resource manager detect a resource under control of an operations center that has no logged in users, the shared resource manager pushes an event to notify others.

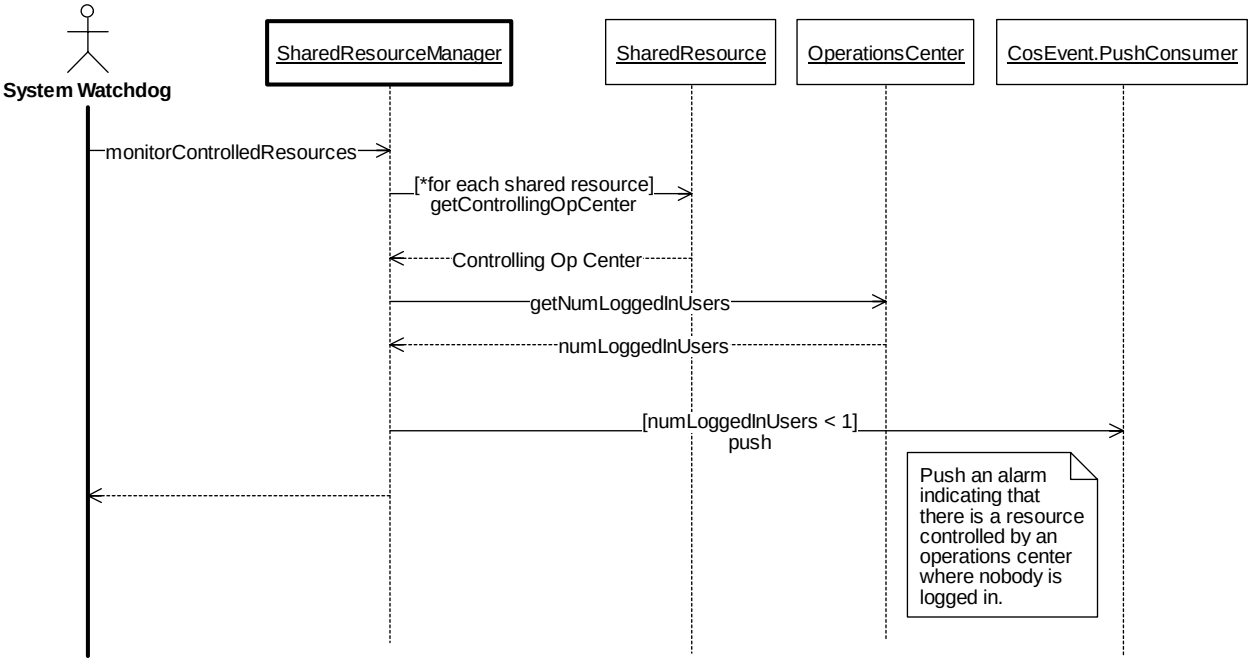

Figure 26. MonitorControlledResources:Basic (Sequence Diagram)

5.25 PollDMS:Basic (Sequence Diagram)

A user with the proper functional rights can poll a DMS for its current status outside of the normal polling cycle. Since this will require field communications which may be time consuming, the command is executed asynchrounously by the DMS and a command status object is used to keep the caller apprised of the execution status. An event is pushed via the CORBA Event Service to notify the caller and others of the new DMS status following the poll attempt.

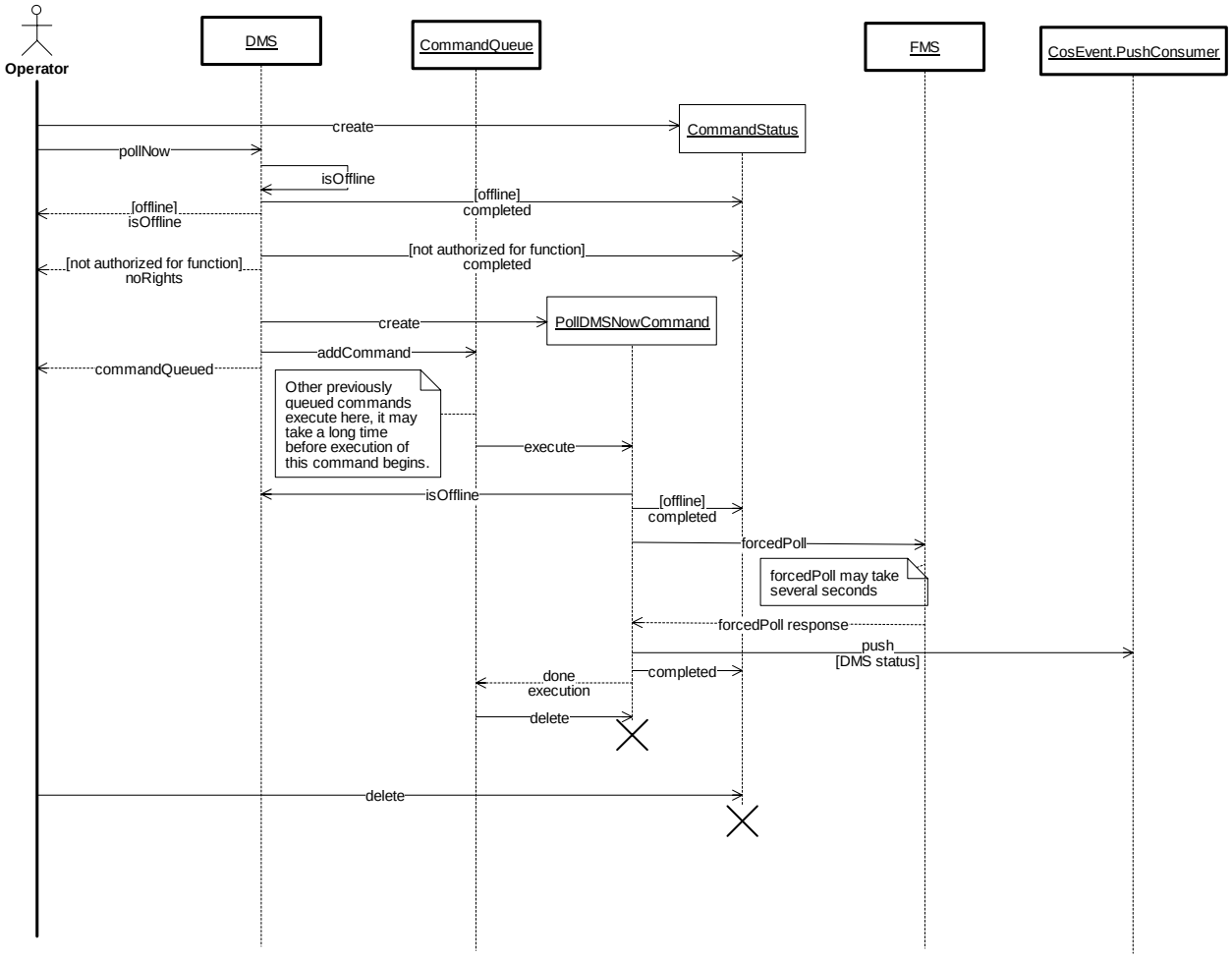

Figure 27. PollDMS:Basic (Sequence Diagram)

5.26 RemoveBannedWord:Basic (Sequence Diagram)

A user with the proper functional rights may remove banned words from the dictionary used to validate messages destined for a DMS. An event is pushed following the update of the dictionary to notify those that may have the dictionary in a local cache.

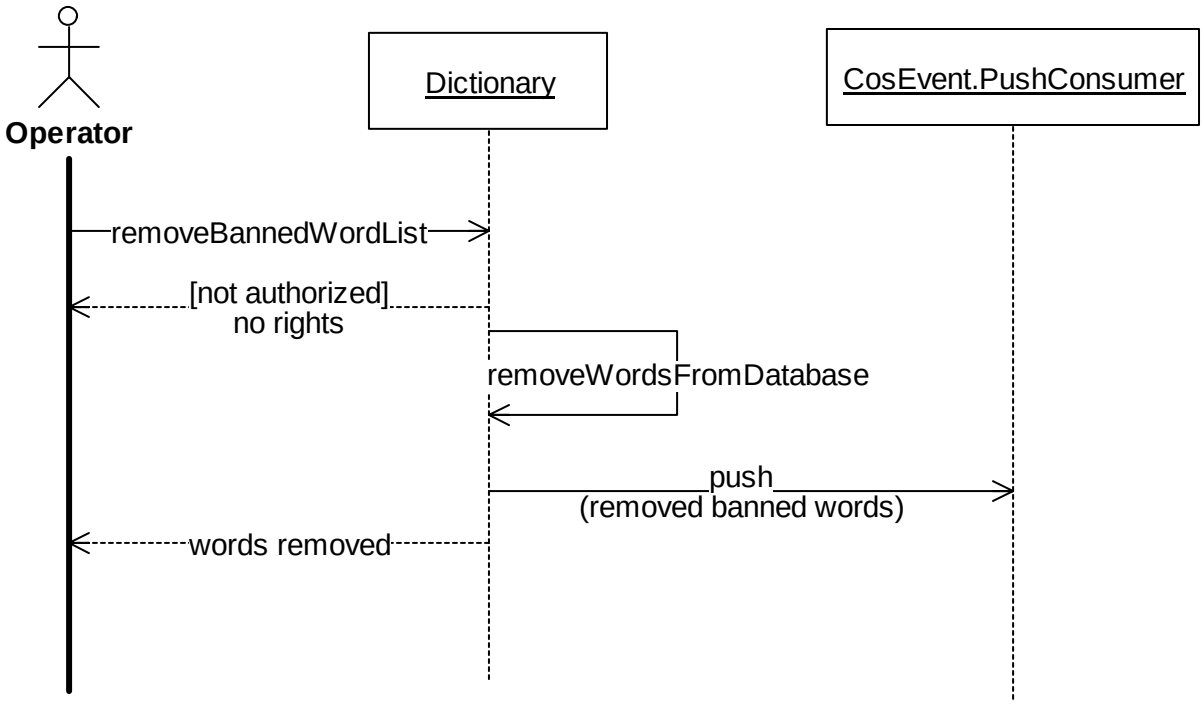

Figure 28. RemoveBannedWord:Basic (Sequence Diagram)

5.27 ResetDMS:Basic (Sequence Diagram)

A user with the proper functional rights may reset a DMS controller. Since this operation involves field communications, the operation is carried out asynchronously by the DMS and a command status object is used by the caller to be kept apprised of the progress of the operation. The DMS is blanked prior to the reset to insure all DMSs that are reset are put into a consistent state. All rules governing shared resources apply to a reset operation, which involves checking the controlling operations center of the device against the operations center of the user prior to allowing the operation to be performed.

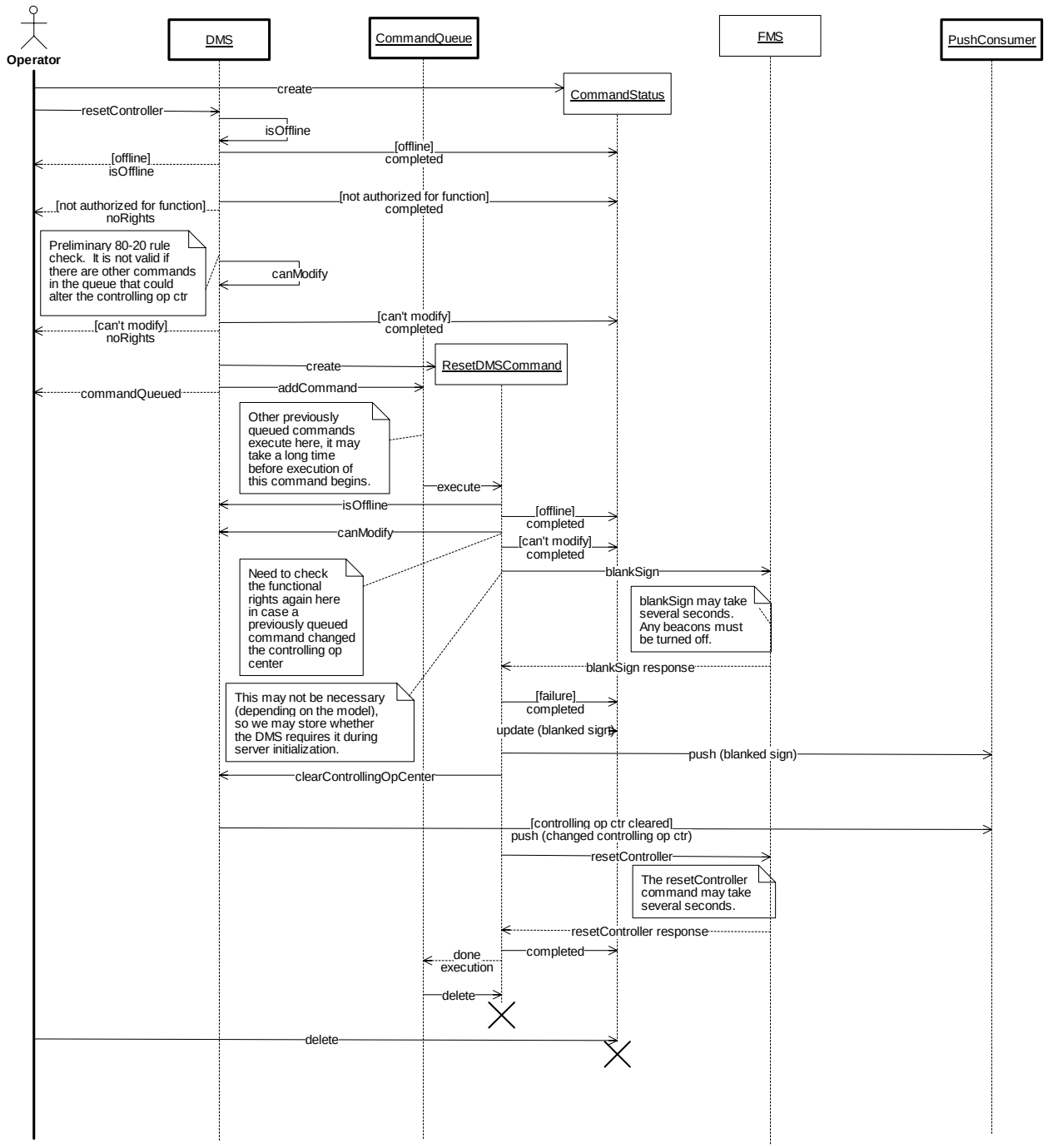

Figure 29. ResetDMS:Basic (Sequence Diagram)

5.28 RevokeRole:Basic (Sequence Diagram)

A user with the proper functional rights may revoke a role from another user. This modifies the functional rights of the user and takes effect upon the next time the user logs into the system.

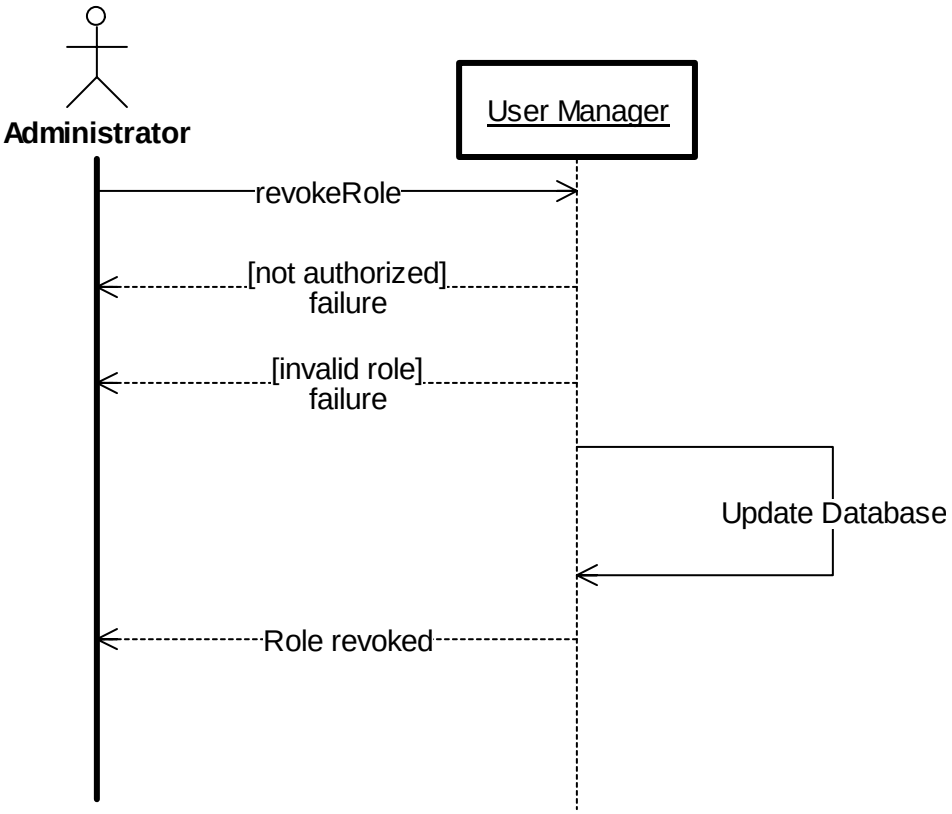

Figure 30. RevokeRole:Basic (Sequence Diagram)

5.29 SetDMSLibraryName:Basic (Sequence Diagram)

A user with the proper functional rights may set the name assigned to a DMS stored message library. An event is pushed via the CORBA Event Service to notify others of the name change.

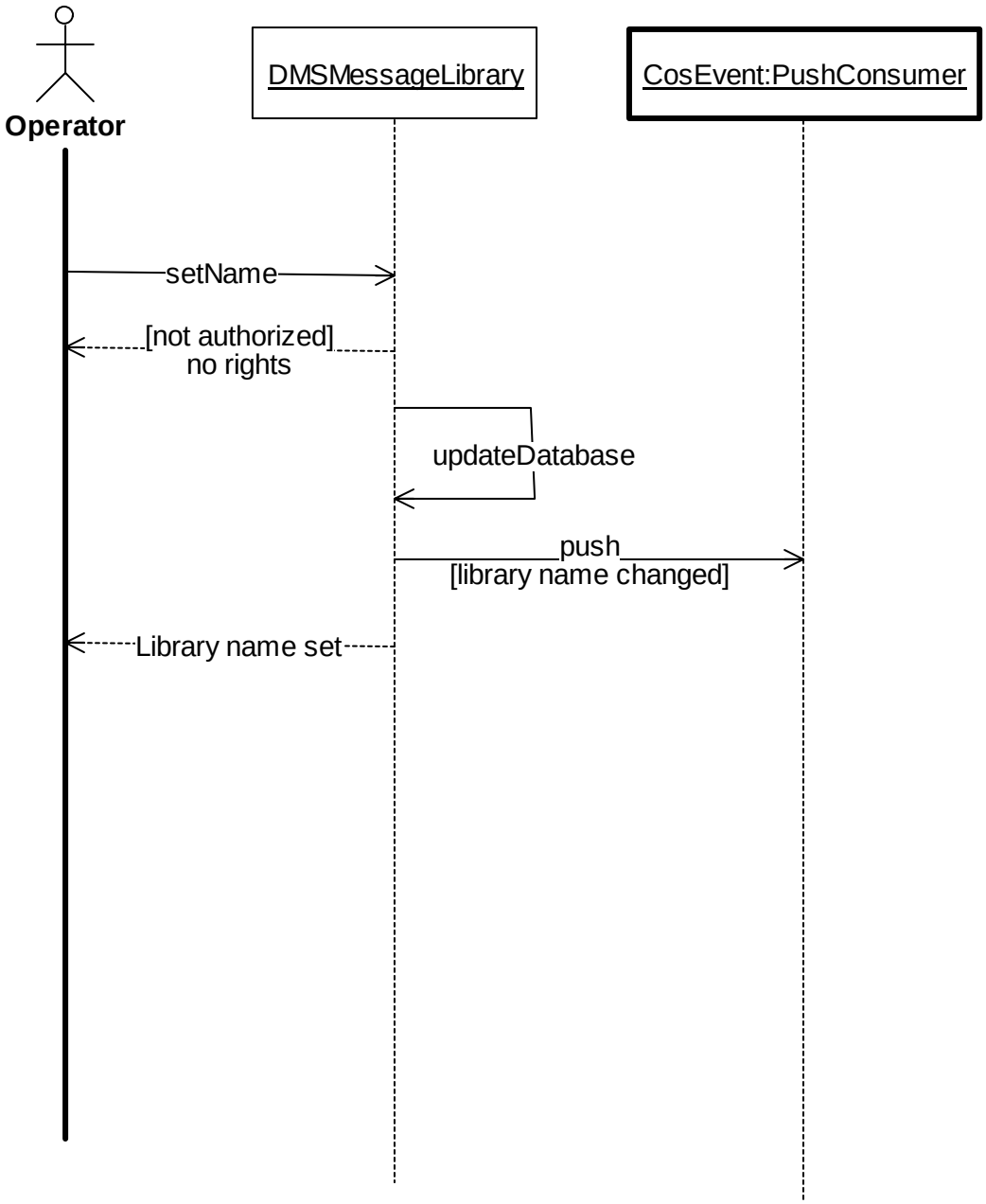

Figure 31. SetDMSLibraryName:Basic (Sequence Diagram)

5.30 SetDMSMessage:Basic (Sequence Diagram)

A user possessing the proper functional rights may set the display (including beacons) of a DMS. DMS objects that are offline cannot be communicated with. Setting the message on a DMS involves the concept of shared resource management, allowing only the controlling operations center to set the message on a DMS that is not blank (unless the user has the override functional right). Once clear to perform the operation, the command is queued within the DMS object and the user is notified that a long running operation is in progress. The supplied CommandStatus object is used to notify the caller of the ongoing progress. When the DMS pulls the command off its internal queue, it executes the command using the FMS subsystem. If the command is successful, the controlling operations center is set . The CORBA event service is used to push state changes of the DMS, for both the action of the message being set on the sign and the controlling operations center being set.

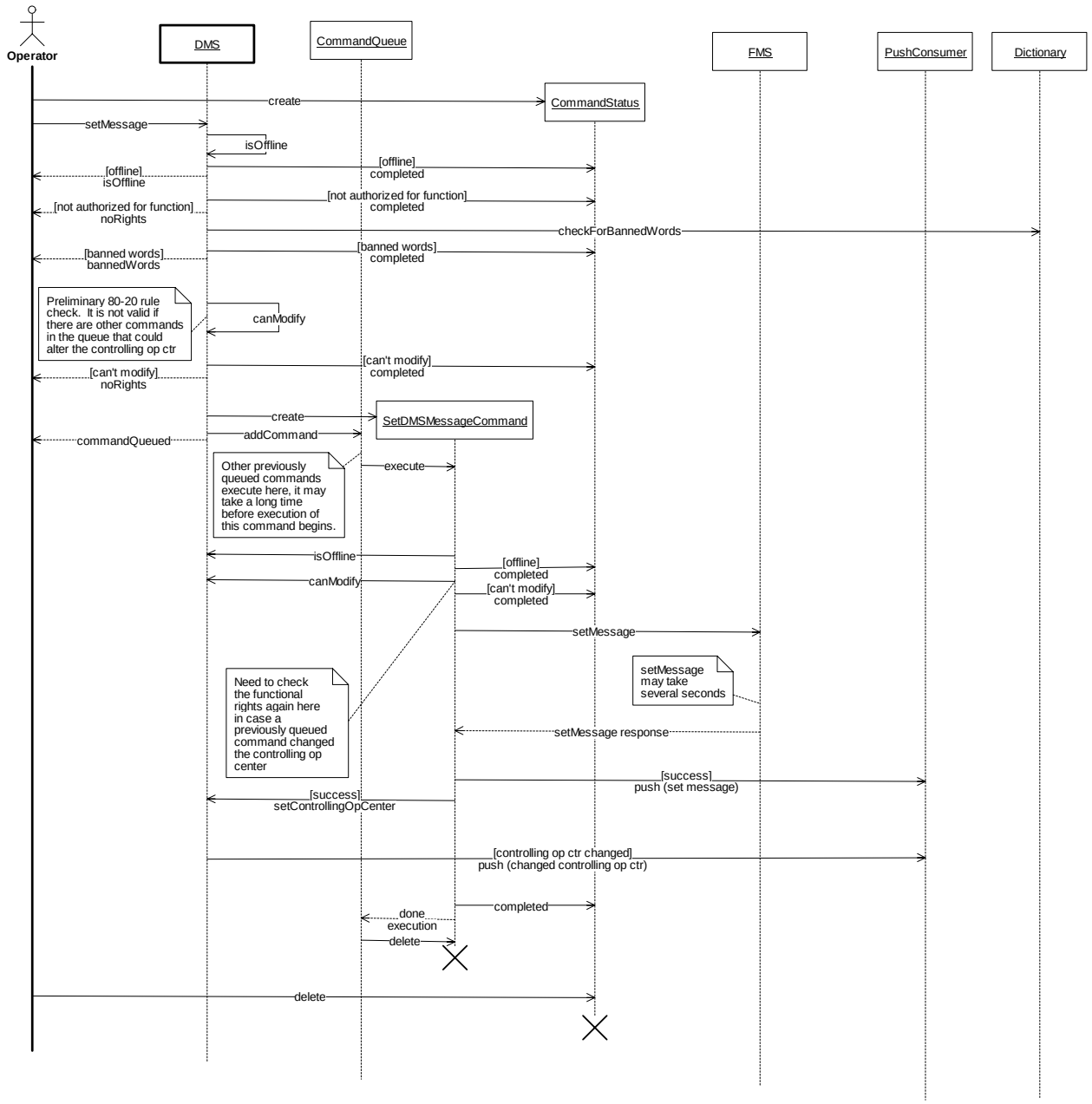

Figure 32. SetDMSMessage:Basic (Sequence Diagram)

5.31 SetDMSName:Basic (Sequence Diagram)

A user with the proper functional rights can change the name of a DMS. An event is pushed via the CORBA Event Service to notify others of the name change.

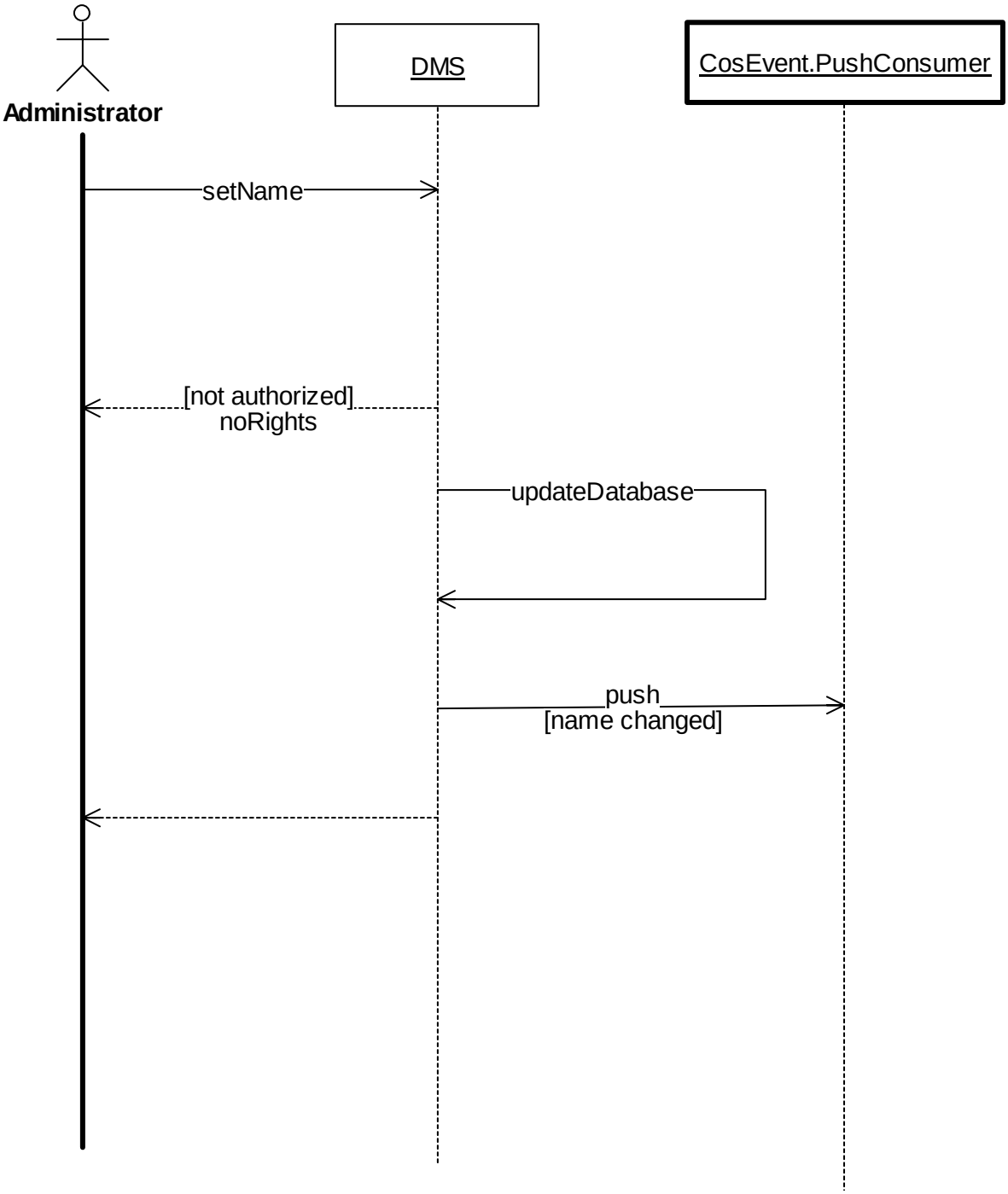

Figure 33. SetDMSName:Basic (Sequence Diagram)

5.32 SetDMSOffline:Basic (Sequence Diagram)

A user with the proper functional rights can set a DMS offline if the DMS is blank or failed. Taking a DMS offline involves FMS communications and may take an extended amount of time. For this reason, the operation is executed asynchronously and a command status object is used to keep the caller informed of the execution status. An attempt is made to blank the DMS before taking it offline. Taking the DMS offline has the effect of stopping automatic polling and disallows any further operations other than to put the DMS online. Shared resource management rules apply to this operation. If the DMS is under the control of an operations center, only a user from that operations center or a user with override functional rights may take the DMS offline. Taking the DMS offline clears the controlling operations center.

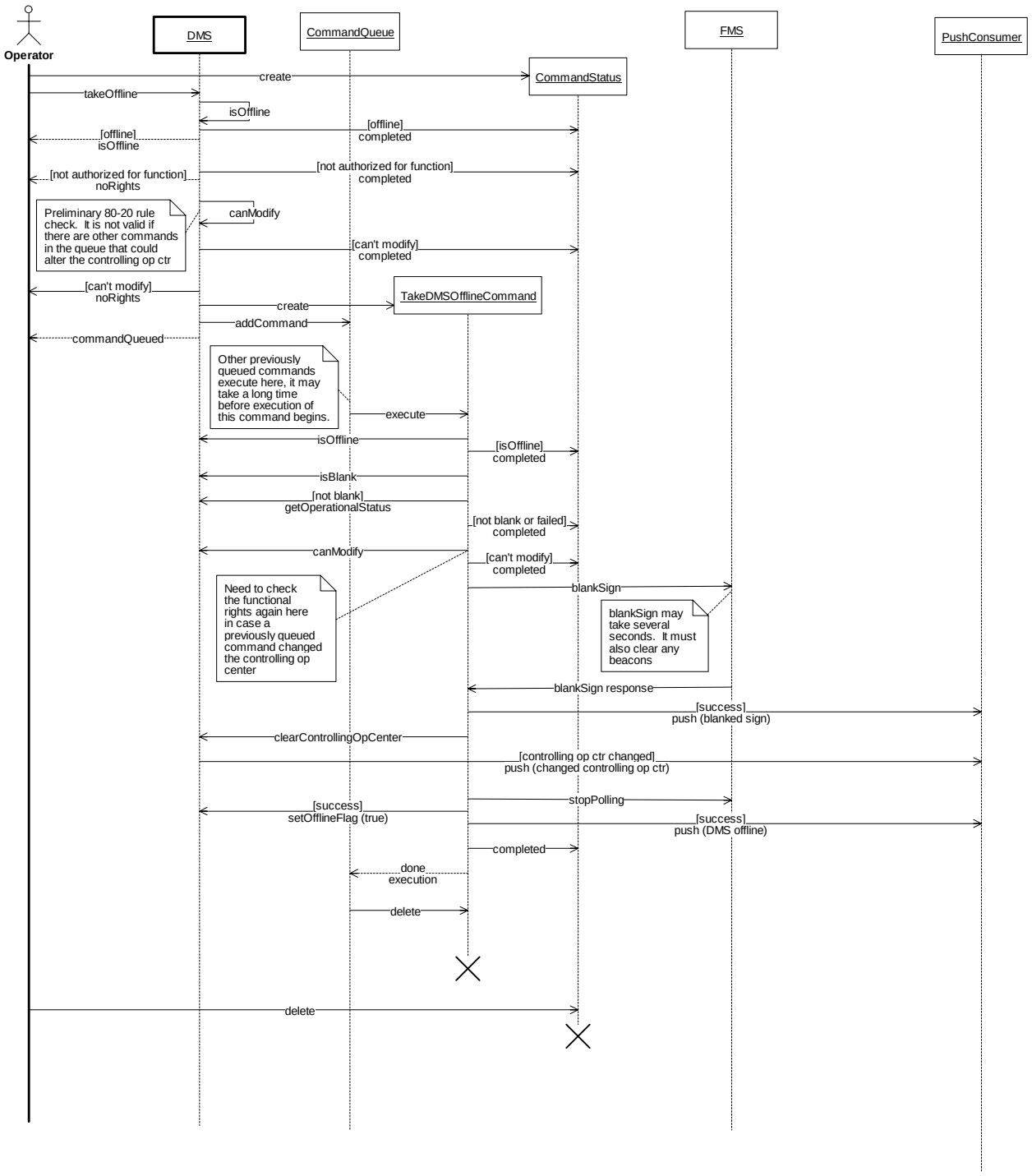

Figure 34. SetDMSOffline:Basic (Sequence Diagram)

5.33 SetDMSOnline:Basic (Sequence Diagram)

A user with the proper functional rights may put a DMS online. Putting the DMS online involves FMS communications and is therefore done asynchronously. A command status object is used by the caller to monitor the status of the operation. Before a DMS is brought online, it is blanked to insure its status is consistent with the status known by the system. Automatic polling of the DMS is started within the FMS subsystem.

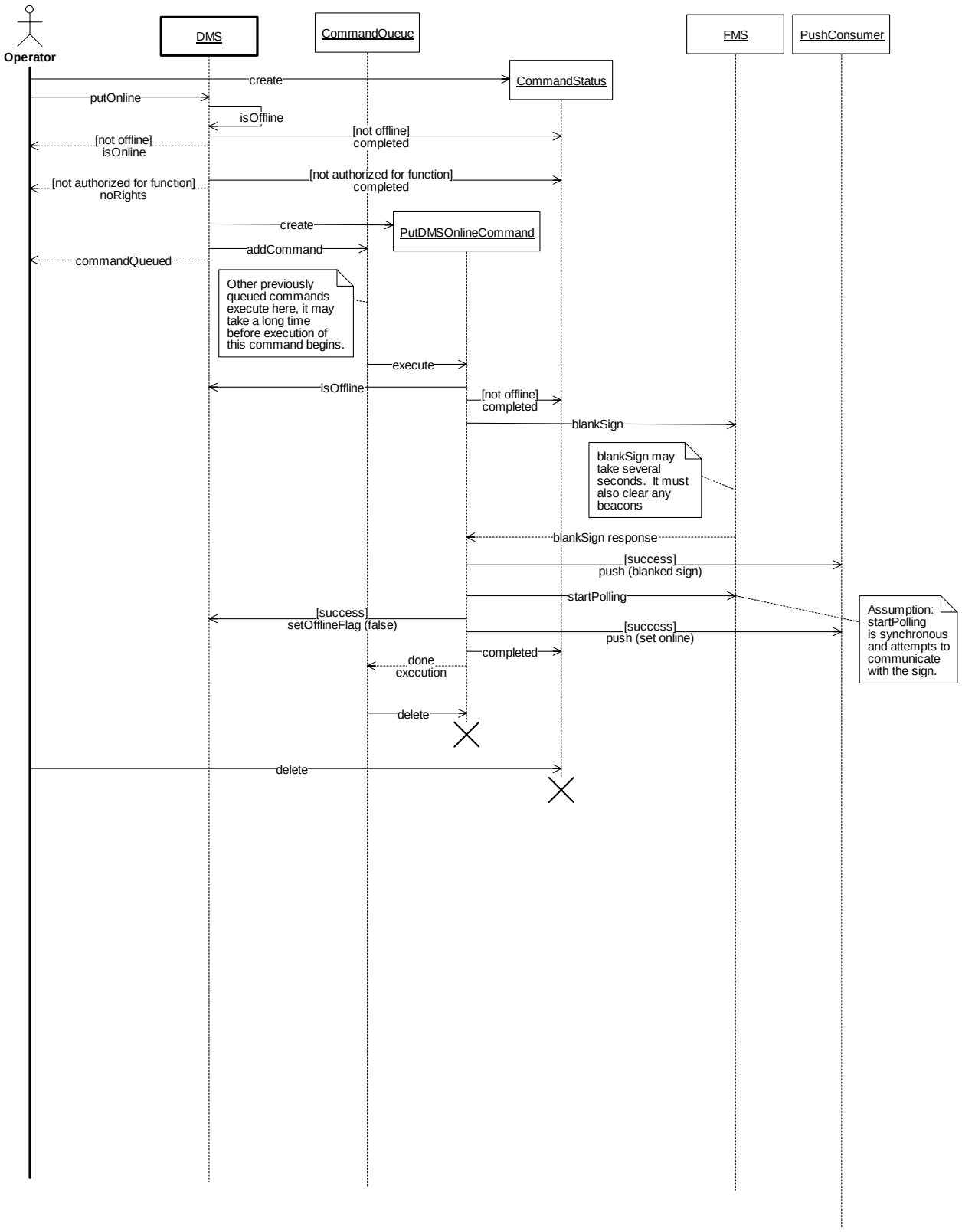

Figure 35. SetDMSOnline:Basic (Sequence Diagram)

5.34 SetDMSPollingInterval:Basic (Sequence Diagram)

A user with the proper functional rights can set the polling interval used by the FMS subsystem to monitor the DMS. Since FMS communications are required, the operation is performed asynchronously and a command status is used by the caller to monitor the progress of the command.

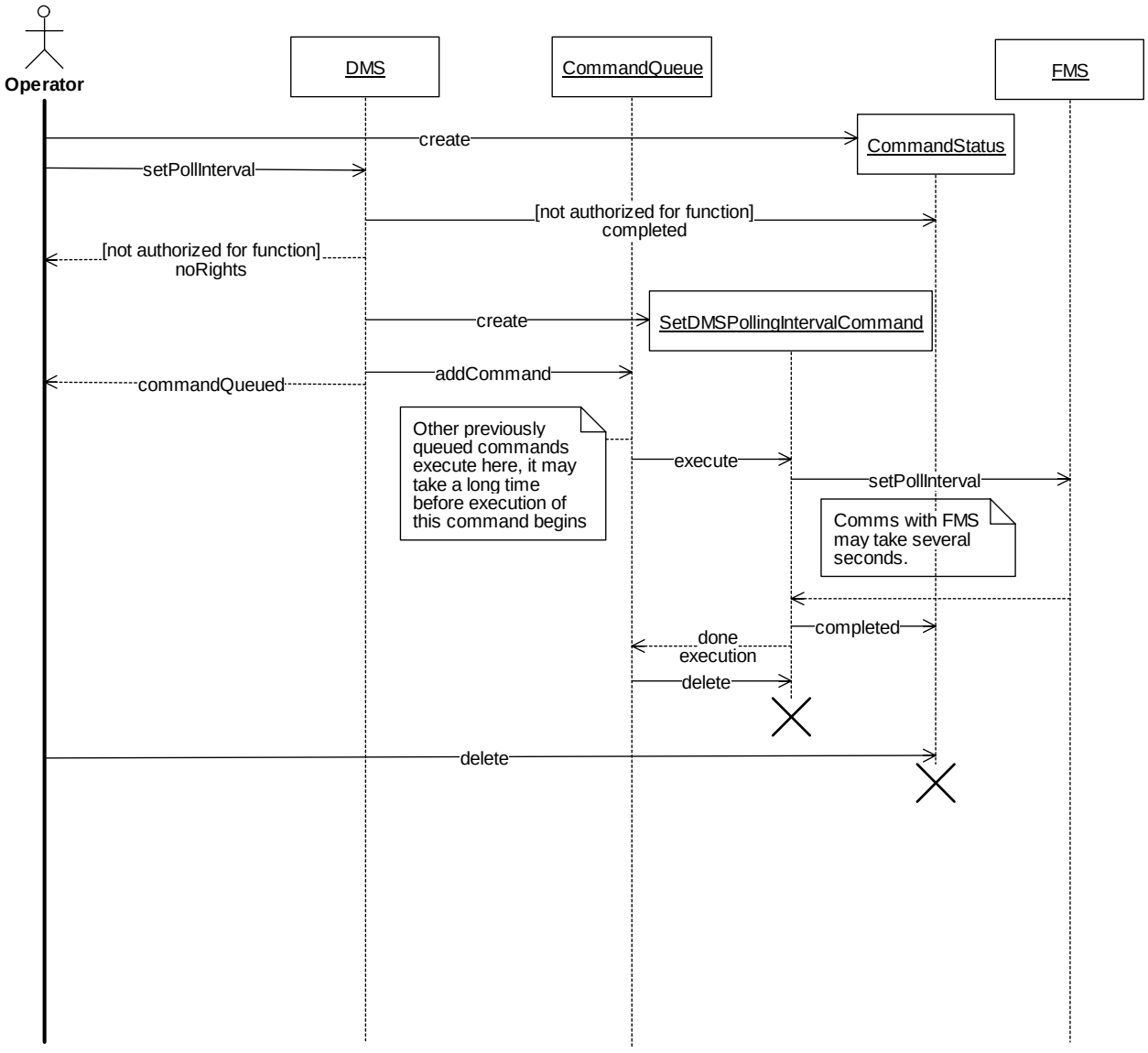

Figure 36. SetDMSPollingInterval:Basic (Sequence Diagram)

5.35 TransferResponsibility:Basic (Sequence Diagram)

A user with the proper functional rights can move the control of a shared resource from one operations center to another. The target operations center must have users logged in to be able to be used as the target of this operation. The change to the shared resource's controlling operations center is pushed as an event via the CORBA Event Service to notify others of the change.

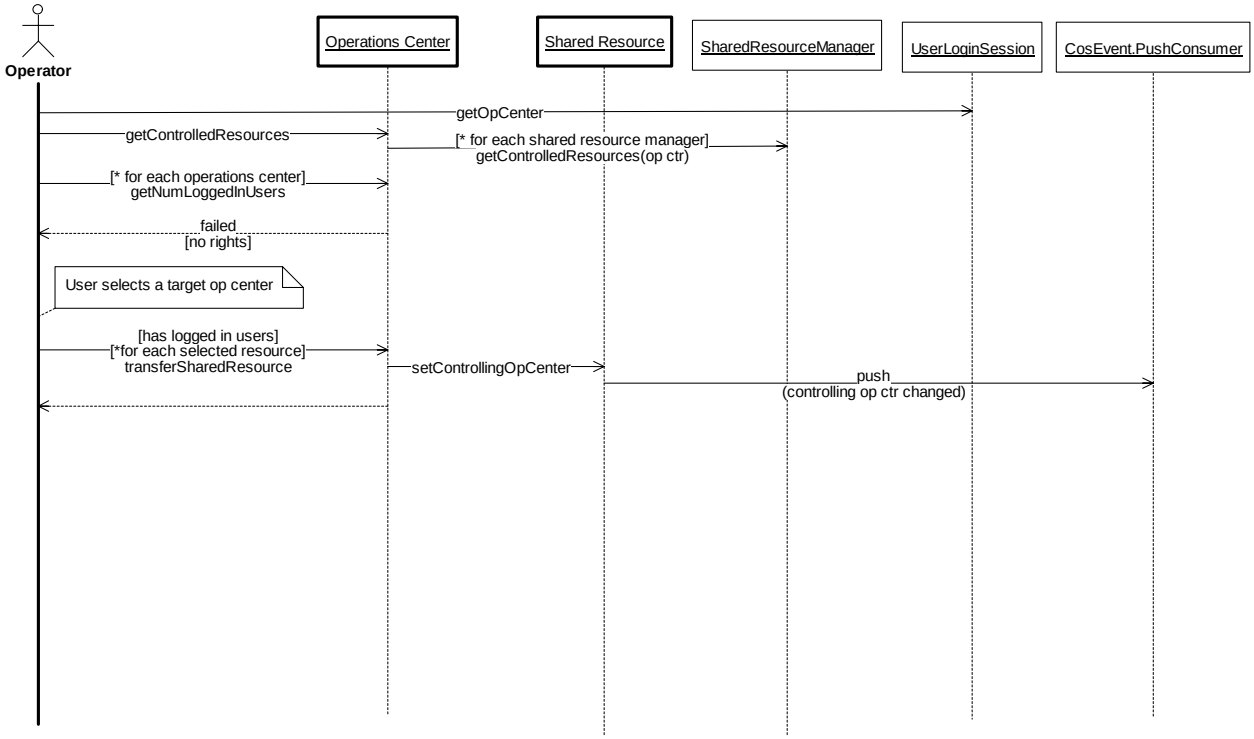

Figure 37. TransferResponsibility:Basic (Sequence Diagram)

5.36 ViewDMSStatus:Basic (Sequence Diagram)

This sequence diagram depicts the fact that the status of a DMS may be obtained on demand, or can be received periodically as the state of the DMS changes. State changes are pushed as events via the CORBA Event Service.

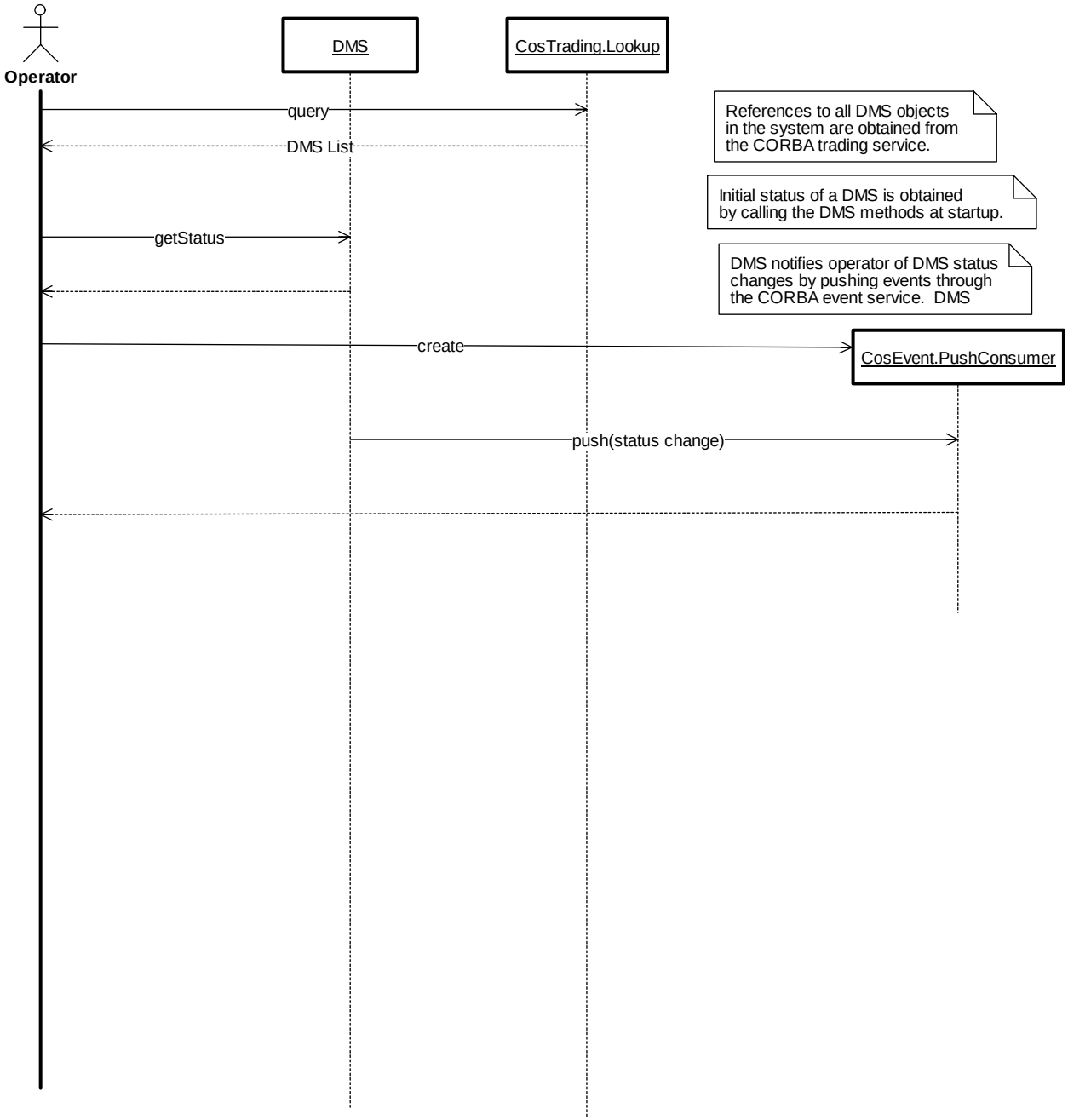

Figure 38. ViewDMSStatus:Basic (Sequence Diagram)

6 – Packaging

Packaging is a way of breaking up a large system into more manageable pieces. Each package has a well-defined set of interfaces. Changes may occur within the internals of a package without affecting the rest of the system. Changes to an interface to a package may require changes to other packages, as shown with dependencies. The diagram below shows the packaging of the Chart II Release 1 Build 1 system.

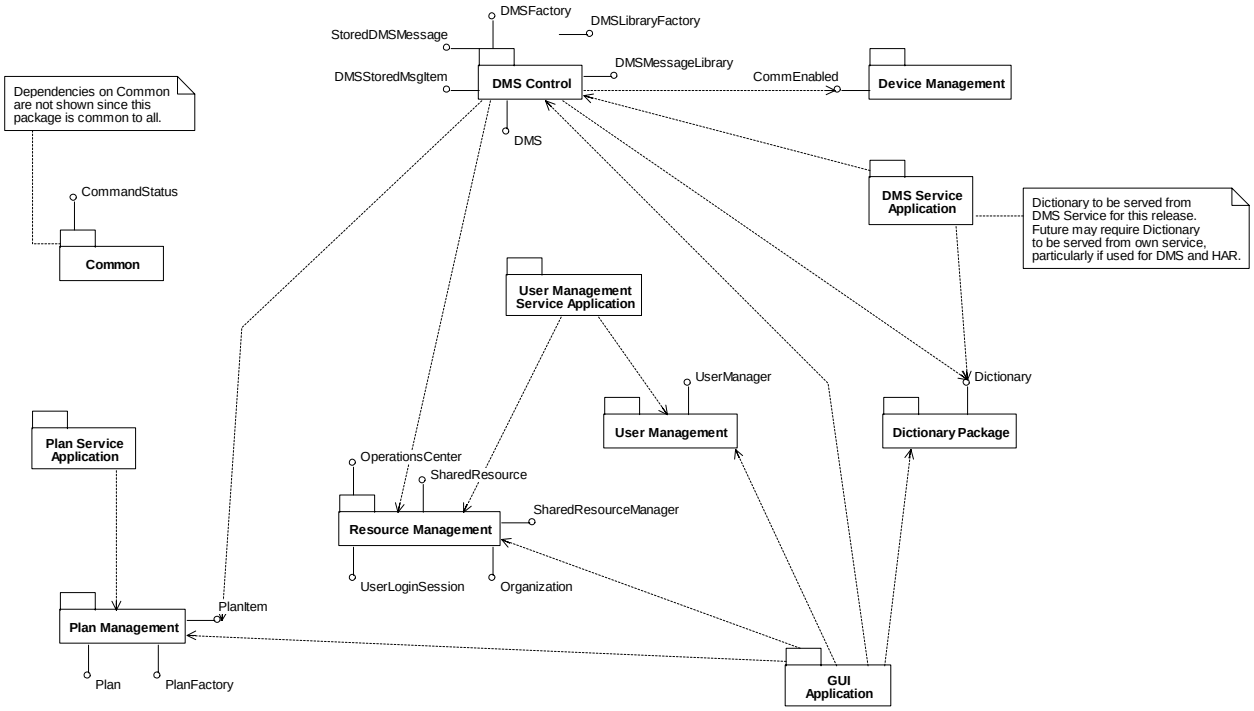

Figure 39. R1B1PackageDiagram (Class Diagram)

This deployment diagram shows which applications will serve each of the CORBA interfaces in the system. Dependencies are used to show where applications require another application for full functionality. Since the nature of the Chart II system is that partial functionality will be available even in the event of a system failure, a note on each dependency discusses why the dependency exists and the functionality that is dependent upon it. Note that lines showing dependencies on the CORBA services are not shown explicitly. A note attached to each of these services describes the dependencies in detail.

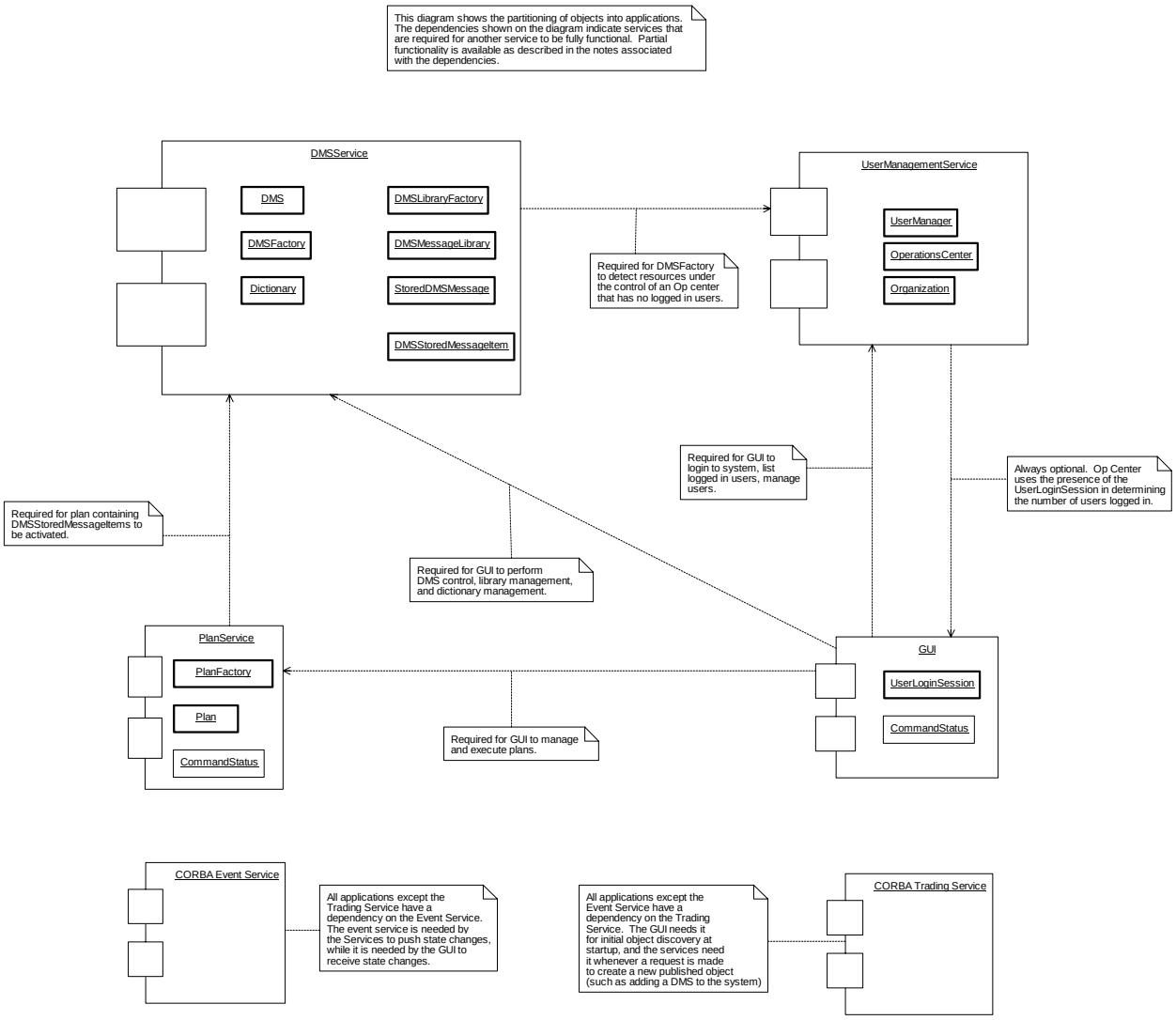

Figure 40. R1B1DeploymentDiagram (Deployment Diagram)

7 - Deployment

This deployment diagram shows where each of the Chart II executables will be run for Release 1 Build 1. In addition to the Chart II applications, the existing AVCM applications are shown to emphasize the coexistence of these two applications.

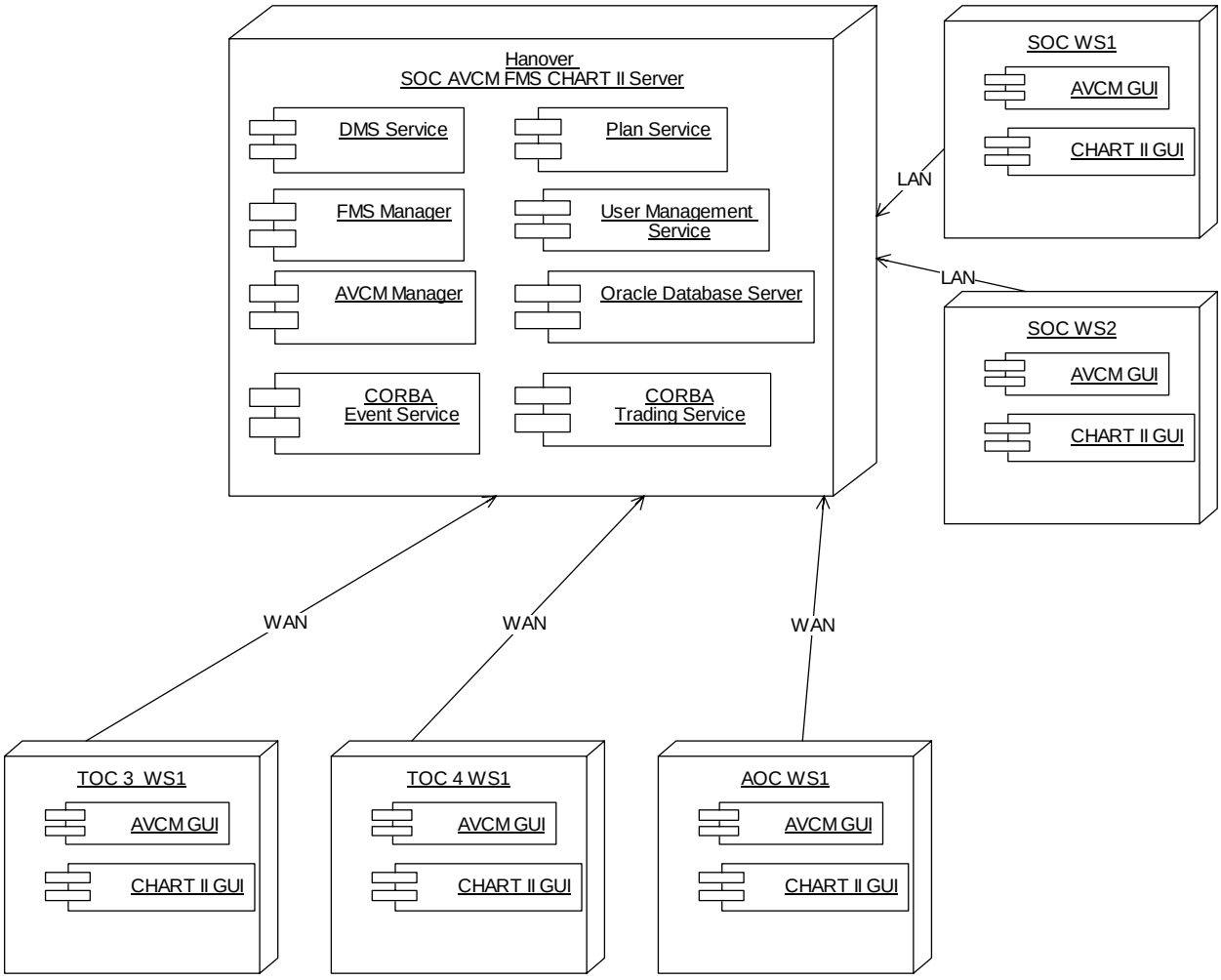

Figure 41. R1b1 (Deployment Diagram)

8 – Interface Definition Language (IDL)

This section lists the IDL that is defined for the CORBA objects for release 1 build 1. The IDL is laden with comments that are able to be parsed by an automatic document generator. The generated documentation is available in the IDL directory accompanying this paper. This online format may be more desirable for review, however the hardcopy is contained here.

```
/*

File Name    : Common.idl

Prepared By  : Computer Sciences Corporation / PB Farradyne, Inc.

+------+   Copyright 1999, Maryland Department of Transportation
| MDOT |                   All Rights Reserved
+------+-----------------------------------------------------------
This file is property of the Maryland Department of
Transportation.  Any unauthorized use or duplication of this
file, or removal of the header, constitutes theft of
intellectual property.

Description  : This file contains contains definitions needed by
               multiple modules in the CHART 2 project.

History
Date        Author         Description
--------------------------------------------------------------------------------
-

*/

#ifndef _COMMON_IDL_
#define _COMMON_IDL_

#pragma prefix "CHART2"

/**
* This module contains definitions needed by multiple modules.
*/
module Common
{
   /**
   * An access token is an internally generated byte sequence
   * which contains at least all of the permissions given to
   * the user.  The token is generated by the call to LoginUser()
   * and must be passed to all methods which are restricted.
   * Restricted methods include those which could negatively
   * impact the operation of the system, as well as those which
   * return sensitive information.
   */
   typedef sequence<octet> AccessToken;
```

```
   /**
   * User name of a CHART2 user.
   * Typedef is used to protect the IDL from future changes.
   */
   typedef string UserName;


   /**
   * Password of a CHART2 user.
   * Typedef is used to protect the IDL from future changes.
   */
   typedef string Password;


   /**
   * A network connection site is the site on the network where an object is
being served.
   * Typically this will be the name of the district whose server is running
a service.
   */
   typedef string NetworkConnectionSite;


   /**
   * Generic exception class for the CHART2 system.
   *
   * This class can be used for throwing very generic exceptions
   * which require no special processing by the client.  Is supports
   * a reason string which may be shown to any user and a debug string
   * which will contain detailed information useful in determining the
   * cause of the problem.
   *
   * @member reason - High level error description
   * @member debug - Detailed information to help resolve the problem
   */
   exception CHART2Exception
   {
      string reason;
      string debug;
   };


   /**
   * Exception which describes an access denied, or "no rights" failure.
   *
   * @member reason - the explanation for the failure.
   */
   exception AccessDenied
   {
      string reason;
      string requiredRights;
   };
```

```
   /**
    * Internal identifier type for all public objects.
    * Composed of the following elements
    *  - 4 byte sequence id which must be unique only within this second
    *  - 4 byte time object was created in seconds from epoch
    *  - 2 byte port of service where object is originally served
    *  - 4 byte ip_address of service where object is originally served
    */
   typedef sequence<octet> Identifier;


   /**
   * Interface for client updates on long running operations.
   *
   * This interface is used to allow the initiating client to
   * track the status of an operation which is long running and
   * thus, performed asynchronously by a server.  The client may create
   * a command status object and pass it to the operation.  The server
   * will then invoke the update and completed methods to notify the
   * client of significant events.  Refer to the definition of the
   * specific long running operation to determine if the CommandStatus
   * object is optional.
   */
   interface CommandStatus
   {
      /**
      * Method to update the current status of the command.
      *
      * This method may be called repeatedly by the server to notify
      * the client of the current status of a command.
      *
      * @param status - Description of the current status.
      */
      void update(in string status);

      /**
      * Method to notify the client that the command has completed.
      *
      * This method may be called only once by the server to notify the
      * client that the command has completed, and send final status
      * information.  After this method is invoked, the client is free to
      * dispose of this object.
      *
      * @param commandSuccessful - Indicates if the command completed
      *                            successfully or not.
      * @param finalStatus - Description of the final command status.
      */
      void completed(in boolean commandSuccessful, in string finalStatus);

   };  // end interface CommandStatus
};

#endif


/*
```

```
File Name      :  DeviceManagement.idl

Prepared by    :  Computer Sciences Corporation / PB Farradyne, Inc.

+------+   Copyright 1999, Maryland Department of Transportation
| MDOT |                   All Rights Reserved
+------+-----------------------------------------------------------
This file is property of the Maryland Department of
Transportation.  Any unauthorized use or duplication of this
file, or removal of the header, constitutes theft of
intellectual property.

Description    :  This file contains the IDL definitions that could
                  possibly apply to more than one type of device.

History
Date        Author                  Description
--------------------------------------------------------------------------------
-
*/


#ifndef _DEVICE_MANAGEMENT_IDL_
#define _DEVICE_MANAGEMENT_IDL_

#include <Common.idl>

#pragma prefix "CHART2"


/**
* This module contains the definitions of the IDL interfaces
* that may be common to more than one type of device.
*/
module DeviceManagement
{
   /**
   * This enum lists the values that can be used to describe the operational
status of a
   * CHART2 system device.
   *
   * @member OK               The DMS is working properly.
   * @member COMM_FAILURE     There has been a failure communicating to the
DMS.
   * @member HARDWARE_FAILURE The DMS is reporting a hardware failure.
   *
   */
   enum OperationalStatus
   {
      OK,
      COMM_FAILURE,
      HARDWARE_FAILURE
   };

   /**
```

```
   * Event information for a controlling op center change.
   *
   * @member resourceID ID of the resource for which the controlling
operations center has been changed.
   * @member opCtrName Name of the new operations center.
   * @member opCtrName ID of the new operations center.
   */
   struct ControllingOpCtrChangeEventInfo
   {
      Common::Identifier resourceID;
      string             opCtrName;
      Common::Identifier opCtrID;
   };


   /**
   * Interface which is common to all hardware that can be communicated with.
   * When offline, no communication will occur with the hardware.
   * When online, the system may attempt to communicate with the hardware.
   */
   interface CommEnabled
   {
      /**
      * Puts the hardware online so that it can be communicated with.
      *
      * @param token Access token used to verify the caller's right to
perform this operation.
      *
      * @exception Common::AccessDenied Caller does not have proper rights
      *                                 to perform this operation.
      * @exception Common::CHART2Exception A general error occurred while
attempting the
      *                                    command.
      */
      void putOnline(in Common::AccessToken token)
         raises (Common::AccessDenied, Common::CHART2Exception);


      /**
      * Takes the hardware offline so that it cannot be communicated with.
      *
      * @param token Access token used to verify the caller's right to
perform this operation.
      *
      * @exception Common::AccessDenied Caller does not have proper rights
      *                                 to perform this operation.
      * @exception Common::CHART2Exception A general error occurred while
attempting the command.
      */
      void takeOffline(in Common::AccessToken token)
         raises (Common::AccessDenied, Common::CHART2Exception);


      /**
      * Determines whether the hardware is offline or online.
```

```
      *
      * @return True if the hardware is offline, false if it's online.
      */
      boolean isOffline();
   };

};  // end module DeviceManagement

#endif



/*

File Name    : Dictionary.idl

Prepared By  : Computer Sciences Corporation / PB Farradyne, Inc.

+------+   Copyright 1999, Maryland Department of Transportation
| MDOT |                   All Rights Reserved
+------+-----------------------------------------------------------
This file is property of the Maryland Department of
Transportation.  Any unauthorized use or duplication of this
file, or removal of the header, constitutes theft of
intellectual property.

Description  : This file contains interface definition language desribing
               the module containing Dictionary related definitions for
               the CHART 2 project.

History
Date        Author         Description
--------------------------------------------------------------------------------
-

*/

#ifndef _CHART2_DICTIONARY_IDL_
#define _CHART2_DICTIONARY_IDL_

#pragma prefix "CHART2"

#include <Common.idl>


/**
* This module contains interfaces needed for the CHART2 system dictionary.
*/
module Dictionary
{
   /**
   * A list of words
   */
   typedef sequence<string> WordList;
```

```
   /**
   * This enum lists the events related to dictionary.  These events are
   * pushed by the Dictionary service through an untyped channel
   * of a CORBA event service.  The data pushed with these events is
   * defined in the DictionaryEvent union.
   *
   * @member BannedWordsAdded     pushed when banned words are added to the
dictionary.
   * @member BannedWordsRemoved   pushed when banned words are removed from
the dictionary.
   *
   * @see DictionaryEvent
   */
   enum DictionaryEventType
   {
      BannedWordsAdded,
      BannedWordsRemoved
   };

   /**
   * Data that is passed with every banned word event.
   *
   * @member dictionaryID - ID of the dictionary that has been altered.
   * @member string theWord - List of words related to the event.
   */
   struct BannedWordEventData
   {
      Common::Identifier   dictionaryID;
      WordList             listOfWords;
   };


   /**
   * This union identifies the data to be passed with events that are pushed
   * through the CORBA event service.
   *
   * @see DictionaryEventType
   */
   union DictionaryEvent switch(DictionaryEventType)
   {
      case BannedWordsAdded:
      case BannedWordsRemoved:
         BannedWordEventData eventData;
   };


   /**
   * Interface for management and utilization of a system dictionary.
   *
   * This interface provides methods to modify the contents of the
   * CHART2 system dictionary.  It also provides a method to check
   * a delimited string for banned words.  The Dictionary is a list
   * of banned words against which all traveller information messages
   * must be checked before being exposed to the public.
   * (e.g. DMS messages before being sent to a sign)
```

```
   */
   interface Dictionary
   {
      /**
      * Get the unique identifier for this dictionary.
      *
      * @return - The dictionary identifier.
      */
      Common::Identifier getID();

      /**
      * Get the current list of banned words.
      *
      * This method will return the list of banned words currently
      * stored in the dictionary database.  If there are no words in
      * the dictionary, this method will return an empty sequence.
      *
      * @param token - The access token of the invoking client.
      *
      * @return - The current list of banned words.
      *
      * @exception Common::AccessDenied - Thrown when the invoking client
does not have
      *                                   the proper functional rights to
call this
      *                                   method.
      * @exception Common::CHART2Exception - Thrown when a general error
condition
      *                                      keeps this method from
completing
      *                                      successfully.
      */
      WordList getBannedWords(in Common::AccessToken token)
         raises(Common::AccessDenied, Common::CHART2Exception);


      /**
      * Removes a list of banned words from the dictionary.
      *
      * This method will remove the specified words from the current
      * list of banned words in the dictionary.  This method will
      * not complain if any of the specified words are not in the dictionary.
      * Because the desired end-state is achieved.  The words are not in
      * the dictionary.
      *
      * @param token - The access token of the invoking client.
      * @param bannedWordList - The words to remove from the dictionary.
      *
      * @exception Common::AccessDenied - Thrown when the invoking
      *                                   client does not have the
      *                                   proper functional rights to
      *                                   call this method.
      * @exception Common::CHART2Exception - Thrown when a general error
condition
```

```
      *                                       keeps this method from
completing
      *                                       successfully.
      */
      void removeBannedWordList(in Common::AccessToken token, in WordList
bannedWords)
         raises(Common::AccessDenied, Common::CHART2Exception);


      /**
      * Adds a list of banned words to the dictionary.
      *
      * This method will add the specified list of words to the current
      * list of banned words in the dictionary.  This method will
      * not complain if any of the specified words were already in the
      * dictionary.  Because the desired end-state is achieved.  The word
      * is in the dictionary.
      *
      * @param token - The access token of the invoking client.
      * @param bannedWords - The list of words to add to the dictionary.
      *
      * @exception Common::AccessDenied - Thrown when the invoking
      *                                   client does not have the
      *                                   proper functional rights to
      *                                   call this method.
      * @exception Common::CHART2Exception - Thrown when a general error
condition
      *                                       keeps this method from
completing
      *                                       successfully.
      */
      void addBannedWordList(in Common::AccessToken token, in WordList
bannedWords)
         raises(Common::AccessDenied, Common::CHART2Exception);


      /**
      * Checks a set of words for banned words.
      *
      * This method will check the passed string for banned words.  The
string
      * will be decomposed into words by using each character in the
delimiters
      * parameter as a delimiter to mark word breaks.  The method will return
      * an empty list if no banned words are found.  Otherwise, it will
return
      * a list containing all banned words in the message.
      *
      * @param messageToCheck - The message to check for banned words.
      * @param delimiters - The set of characters to treat as delimiters.
      *
      * @exception Common::CHART2Exception - Thrown when a general error
condition
      *                                       keeps this method from
completing
```

```
      *                                        successfully.
      */
      WordList checkForBannedWords(in string messageToCheck, in string
delimeters)
         raises(Common::CHART2Exception);
   };

};

#endif



/*

File Name    : DMSControl.idl

Prepared By  : Computer Sciences Corporation / PB Farradyne, Inc.

+------+   Copyright 1999, Maryland Department of Transportation
| MDOT |                   All Rights Reserved
+------+-----------------------------------------------------------
This file is property of the Maryland Department of
Transportation.  Any unauthorized use or duplication of this
file, or removal of the header, constitutes theft of
intellectual property.

Description  : This file contains interface definition language desribing
               the module containing DMS control related definitions for
               the CHART 2 project.

History
Date        Author         Description
--------------------------------------------------------------------------
-

*/


#ifndef _DMS_IDL_
#define _DMS_IDL_

#include <Common.idl>
#include <DeviceManagement.idl>
#include <PlanManagement.idl>
#include <ResourceManagement.idl>
#include <Dictionary.idl>

#pragma prefix "CHART2"

/**
* This module contains definitions relating to the control of dynamic message
signs (DMS)
*/
module DMSControl
```

```
{
   interface DMS;
   interface StoredDMSMessage;
   interface DMSStoredMessageItem;
   interface DMSMessageLibrary;


   /**
   * A MULTI string is a string that uses a mark-up language specified in the
emerging
   * NTCIP standard used for formatting messages for display on a DMS.  The
DMS requires
   * the message text passed to it to be formatted using this standard.
   */
   typedef string MULTIString;


   /**
   * This enum lists the events related to DMS control that are pushed on a
DMS event
   * channel through the CORBA event service.  The data pushed with these
events is
   * defined in the DMSEvent union
   *
   * @member DMSAdded                     pushed when a DMS is added to the
system.
   * @member DMSDeleted                   pushed when a DMS is removed from
the system.
   * @member DMSOnline                    pushed when a DMS is put online.
   * @member DMSOffline                   pushed when a DMS is taken offline.
   * @member ControllingOpCtrChanged      pushed when the op center in control
of a DMS is changed.
   * @member DMSLibraryAdded              pushed when a library has been added
to the system.
   * @member DMSLibraryDeleted            pushed when a libarary has been
removed from the system.
   * @member DMSStoredMessageAdded        pushed when a stored message has
been added to a library.
   * @member DMSStoredMessageChanged      pushed when the contents of a stored
message have been changed.
   * @member DMSStoredMessageDeleted      pushed when a stored message has
been deleted from the system.
   * @member DMSStoredMessageItemAdded    pushed when a plan item used to set
a DMS Message has been added to a plan.
   * @member DMSStoredMessageItemChanged pushed when a DMS plan item has been
changed.
   * @member DMSBlanked                   pushed when a DMS has been blanked.
   * @member DMSMessageChanged            pushed when the message on a DMS has
been set or changed.
   * @member CurrentDMSStatus             pushed when there is a status change
for the DMS (or a forced poll was done).
   * @member DMSNameChanged               pushed when the name of a DMS is
changed.
   *
   * @see DMSEvent
```

```
   */
   enum DMSEventType
   {
      DMSAdded,
      DMSDeleted,
      DMSOnline,
      DMSOffline,
      ControllingOpCtrChanged,
      DMSLibraryAdded,
      DMSLibraryDeleted,
      DMSStoredMessageAdded,
      DMSStoredMessageChanged,
      DMSStoredMessageDeleted,
      DMSStoredMessageItemAdded,
      DMSStoredMessageItemChanged,
      DMSBlanked,
      DMSMessageChanged,
      CurrentDMSStatus,
      DMSNameChanged
   };


   /**
   * This enum provides the valid values for the sign type of a DMS.  The
sign type
   * defines how characters may be displayed on the sign.  Currently, only
character
   * matrix signs are supported.
   *
   * @member CHAR_MATRIX sign modules display 1 char each with physical
intercharacter
   *                     spacing.  This allows the full width of a module to
be used by
   *                     a character.
   */
   enum SignType
   {
      CHAR_MATRIX
   };


   /**
   * This struct defines the content of messages to be placed on DMSs.  The
text and beacon state
   * are coupled and cannot be set independently.  This struct represents a
subset of the NTCIP
   * dmsMessageEntry specification.
   *
   * @member dmsMessageMultiString The text of the message formatted using
the NTCIP MULTI mark up language.
   * @member dmsMessageBeacon 0 = Beacons OFF, 1 = Beacons ON
   */
   struct MessageContent
   {
      MULTIString  dmsMessageMultiString;
```

```
      octet        dmsMessageBeacon;
   };


   /**
   * This struct defines the content of messages stored in a
DMSMessageLibrary.
   * The text and beacon state are coupled and cannot be set independently.
   *
   * @member messageText The text of the message.
   * @member messageTextIsMulti Indicates if the text in this stored
   *                            message is already formatted into the MULTI
   *                            mark-up language or needs the system to
   *                            format the message using the MD SHA
   *                            algorithm.
   * @member beaconsOn True = Beacons ON, False = Beacons OFF
   */
   struct StoredMessageContent
   {
      string          messageText;
      boolean         messageTextIsMulti;
      boolean         beaconsOn;
   };


   /**
   * This struct defines the data included in the status of a DMS.
   *
   * @member onLine Set to true if the DMS is currently on-line.
   * @member msgContent text being shown on the sign (including MULTI
formatting tags) and the beacon state.
   * @member opStatus The operational status of the sign.
   * @member statusChangeTime Date and time status of the DMS last changed
(seconds since epoch).
   */
   struct Status
   {
      boolean                              onLine;
      MessageContent                       msgContent;
      DeviceManagement::OperationalStatus opStatus;
      unsigned long                        statusChangeTime;
   };


   /**
   * This struct defines the data passed in the DMSStoredMessageChanged event
   *
   * @member storedMsgID ID of the stored message that was changed.
   * @member msgContent Content of the message, including the text and beacon
status.
   */
   struct DMSStoredMessageEventInfo
   {
      Common::Identifier   storedMsgID;
      StoredMessageContent msgContent;
```

```
   };


   /**
   * This struct defines the data passed with a DMSStatus event.
   *
   * @member dmsID ID of the DMS
   * @member status Status of the DMS.
   */
   struct DMSStatusEventInfo
   {
      Common::Identifier dmsID;
      Status             status;
   };


   /**
   * This struct defines data passed with a DMSNameChanged event.
   *
   * @member dmsID ID of the DMS whose name was changed.
   * @member name New name given to the DMS.
   */
   struct DMSNameChangedEventInfo
   {
      Common::Identifier dmsID;
      string             name;
   };


   /**
   * This struct defines data passed with a DMSMessageChanged event.
   *
   * @member dmsID DMS whose message was set or changed.
   * @member msgContent Text being displayed on the DMS and the state of the
DMS beacons.
   */
   struct DMSMessageChangedEventInfo
   {
      Common::Identifier dmsID;
      MessageContent     msgContent;
   };


   /**
   * This struct defines the data passed with a DMSStoredMsgItemChanged
event.
   *
   * @member itemID ID of the plan item that was changed.
   * @member name Name of the plan item.
   * @member dmsID ID of the DMS that the plan item operates on.
   * @member msgContent Content of the stored message to be placed on the
DMS.
   */
   struct DMSStoredMessageItemEventInfo
   {
```

```
      Common::Identifier   itemID;
      string               name;
      Common::Identifier   dmsID;
      StoredMessageContent msgContent;
   };


   /**
   * This union identifies the data to be passed with events that are pushed
through the
   * event service.
   *
   * @see DMSEventType
   */
   union DMSEvent switch (DMSEventType)
   {
      case ControllingOpCtrChanged:
         DeviceManagement::ControllingOpCtrChangeEventInfo opCtrInfo;
      case DMSMessageChanged:
         DMSMessageChangedEventInfo msgChangedInfo;
      case DMSAdded:
         DMS dms;
      case DMSOnline:
      case DMSOffline:
      case DMSBlanked:
      case DMSDeleted:
      case DMSLibraryDeleted:
      case DMSStoredMessageDeleted:
         Common::Identifier id;
      case DMSNameChanged:
         DMSNameChangedEventInfo nameChangeInfo;
      case DMSLibraryAdded:
         DMSMessageLibrary lib;
      case DMSStoredMessageAdded:
         StoredDMSMessage storedMsg;
      case DMSStoredMessageChanged:
         DMSStoredMessageEventInfo storedMsgInfo;
      case DMSStoredMessageItemAdded:
         DMSStoredMessageItem msgItem;
      case DMSStoredMessageItemChanged:
         DMSStoredMessageItemEventInfo storedMsgItemInfo;
      case CurrentDMSStatus:
         DMSStatusEventInfo   statusInfo;
   };


   /**
   * This struct defines the data used to identify the size of a DMS.
   * These data elements are a subset of the VMS configuration objects
   * defined in NTCIP. Using the size of the sign and the size of one
   * character of the sign, one can derive the number of rows (for
   * character matrix and line matrix signs) and the number of columns
   * (for character matrix signs).
   *
   * @member vmsSignHeightPixels Indicates the number of rows of pixels
```

```
   *                               contained on the sign. (0..65535)
   * @member vmsSignWidthPixels  Indicates the number of columns of pixels
   *                               contained on the sign. (0..65535)
   * @member vmsCharacterHeigthPixels Indicates the height of a single
character
   *                                in pixels.  A value of 0 is used to
indicate
   *                                a variable character height (full
matrix) (0..255).
   * @member vmsCharacterWidthPixels Indicates the width of a single
character in
   *                               pixels.  A value of 0 is used to
indicate a variable
   *                               character width (full and line matrix)
(0..255).
   * @member
   */
   struct SignMetrics
   {
      long  vmsSignHeightPixels;
      long  vmsSignWidthPixels;
      short vmsCharacterHeightPixels;
      short vmsCharacterWidthPixels;
   };


   /**
   * This struct defines the data used to identify the font used by a DMS.
   * Currently, only fixed width fonts are supported.
   *
   * @member fontHeight height in pixels of all characters in the font.
(0..255)
   * @member characterWidth width in pixels of a character.  For this
implementation,
   *                        this applies to all characters in the font.
(0..255)
   */
   struct FontMetrics
   {
      short fontHeight;
      short characterWidth;
   };

   /**
   * This struct defines configuration data required when adding a DMS to the
system.
   *
   * @member fmsDeviceID            ID of this device as known to the FMS
subsystem.
   * @member name                   Name of this DMS.
   * @member signType               Type of DMS.
   * @member signMetrics            Size of the DMS.
   * @member fontMetrics            Size of the font used by the DMS.
   * @member pages                  Max number of pages (aka phases)
supported by the DMS.
```

```
   * @member agentHostName           Name of the host where the SNMP agent
(FMS subsystem) serving
   *                                 this device is running.
   * @member SNMPCommunityName       SNMP community name used by the SNMP
manager for this device.
   * @member configurableCommTimeout Set to true if this device allows the
comm timeout (keep alive) to be set.
   */
   struct Configuration
   {
      long                  fmsDeviceID;
      string                name;
      SignType              signType;
      SignMetrics           signMetrics;
      FontMetrics           fontMetrics;
      long                  pages;
      string                agentHostName;
      string                SNMPCommunityName;
      boolean               configurableCommTimeout;
   };


   /**
   * This exception is used to indicate that message text for the sign
formatted using the MULTI language
   * is not properly formed and a parse failure occurred.
   *
   * @member reason Reason for the failure.
   */
   exception MULTIParseFailure
   {
      string reason;
   };


   /**
   * This exception is used to indicate that the message content is not
approved.  This could
   * occur if the message text includes banned words or if the beacons are
set ON when the
   * message text is blank.
   *
   * @member reason Reason for the failure.
   */
   exception DisapprovedMessageContent
   {
      Dictionary::WordList disapprovedWords;
      string               reason;
   };


   /**
   * This exception is used to indicate that a particular DMS does not
support the
```

```
   * operation requested.  This typically occurs when the DMS does not
support a feature
   * that is part of the CHART2 feature set.
   *
   * @member reason Reason for the failure.
   */
   exception UnsupportedOperation
   {
      string reason;
   };


   /**
   * This interface is used to represent a DMS in the field.  The system
contains
   * an instance of this interface for each DMS.
   */
   interface DMS : ResourceManagement::SharedResource,
DeviceManagement::CommEnabled
   {
      /**
      * This operation returns the type of this DMS.
      * @return The type of this DMS.
      */
      SignType getSignType();


      /**
      * This operation returns the name that has been given to this DMS.
      *
      * @return The name of this DMS.
      */
      string getName();


      /**
      * This operation sets the name of this DMS.
      *
      * @param token Access token used to verify the caller's right to
perform this operation on this DMS.
      * @param name Name to be assigned to this DMS.
      *
      * @exception Common::AccessDenied Caller does not have proper rights to
perform this
      *                                  operation on this DMS.
      * @exception Common::CHART2Exception An error occurred while setting
the name.
      */
      void setName(in Common::AccessToken token, in string name)
         raises (Common::AccessDenied, Common::CHART2Exception);


      /**
      * This operation returns the current text and beacon status being
displayed by the sign.
```

```
      *
      * @return Current text and beacon state being displayed by the sign.
      */
      MessageContent getMessage();


      /**
      * This operation sets the message and beacon state to be displayed by
the sign.
      * This operation executes asynchronously.  It returns to the caller
after queuing the
      * command for execution.  A CommandStatus object must be used if the
caller wishes
      * to track the progress of the operation.  This operation will trigger
a DMSMessageChanged event
      * when the message has actually been changed on the DMS.
      *
      * @param token Access token used to verify the caller's right to
perform this operation on this DMS.
      * @param msgContent message to be displayed on the DMS.
      * @param status CommandStatus object used to track the progress of this
operation (may be null).
      *
      * @exception Common::AccessDenied Caller does not have proper rights to
perform this
      *                                    operation on this DMS.
      * @exception MULTIParseFailure The message content contains a malformed
MULTI string.
      * @exception DisapprovedMessageContent The content of the message
contains banned words
      *                                        or invalid beacon state.
      * @exception ResourceManagement::ResourceControlConflict Used to
indicate that the resource is already
      *                                                              in use by
another op center.
      * @exception Common::CHART2Exception An error occurred while setting
the message / beacon state.
      */
      void setMessage(in Common::AccessToken token, in MessageContent
msgContent,
                      in Common::CommandStatus status)
         raises (Common::AccessDenied, MULTIParseFailure,
DisapprovedMessageContent,
                 ResourceManagement::ResourceControlConflict,
Common::CHART2Exception);


      /**
      * This operation returns true if the DMS is currently blank.
      *
      * @return True if the DMS is blank, False if the DMS is displaying a
message.
      */
      boolean isBlank();
```

```
      /**
      * This operation blanks the DMS, causing it to stop displaying any
message that it
      * may currently be displaying.
      * This operation executes asynchronously.  It returns to the caller
after queuing the
      * command for execution.  A CommandStatus object must be used if the
caller wishes
      * to track the progress of the operation.  This operation will trigger
a DMSBlanked event
      * when the DMS has actually been blanked.
      *
      * @param token Access token used to verify the caller's right to
perform this operation on this DMS.
      * @param status CommandStatus object used to track the progress of this
operation (may be null).
      *
      * @exception Common::AccessDenied Caller does not have proper rights to
perform this
      *                                    operation on this DMS.
      * @exception ResourceManagement::ResourceControlConflict Used to
indicate that the resource is already
      *                                                          in use by
another op center.
      * @exception Common::CHART2Exception An error occurred while performing
this operation.
      */
      void blankSign(in Common::AccessToken token, in Common::CommandStatus
status)
         raises (Common::AccessDenied,
ResourceManagement::ResourceControlConflict, Common::CHART2Exception);


      /**
      * This operation returns the polling interval used to check the health
of the DMS.
      *
      * @return polling interval in seconds.
      */
      long getPollInterval();


      /**
      * This operation returns the maximum polling interval allowed for this
DMS.  This
      * max polling interval is used to keep the polling interval from being
set larger
      * than the CommLossTimeout for the sign.
      *
      * @return maximum polling interval allowed for this sign (in seconds).
      */
      long getMaxPollInterval();
```

```
      /**
      * This operation is used to set the polling interval for this DMS.
      * This operation executes asynchronously.  It returns to the caller
after queuing the
      * command for execution.  A CommandStatus object must be used if the
caller wishes
      * to track the progress of the operation.
      *
      * @param token Access token used to verify the caller's right to
perform this operation on this DMS.
      * @param interval Polling interval in seconds.  Zero indicates no
polling.
      * @param status CommandStatus object used to track the progress of this
operation (may be null).
      *
      * @exception Common::AccessDenied Caller does not have proper rights to
perform this
      *                                    operation on this DMS.
      * @exception Common::CHART2Exception An error occurred while performing
this operation.
      */
      void setPollInterval(in Common::AccessToken token, in long interval, in
Common::CommandStatus status)
         raises (Common::AccessDenied, Common::CHART2Exception);


      /**
      * This operation returns the date and time the status of this DMS
changed or it was manually polled.
      *
      * @return date and time the status of this DMS last changed.  (seconds
since epoch).
      */
      unsigned long getStatusChangeTime();


      /**
      * This operation returns the current status of the DMS.  This operation
does not cause
      * communications with the sign and instead the last known status is
returned.
      *
      * @return Status of the DMS.
      */
      Status getStatus();


      /**
      * This method causes the DMS to be polled now instead of waiting for
the normal polling
      * cycle.  This operation executes asynchronously.  It returns to the
caller after queuing the
      * command for execution.  A CommandStatus object must be used if the
caller wishes
```

```
      * to track the progress of the operation.  After the sign is contacted,
a CurrentDMSStatus
      * event will be pushed through the event service DMS channel.
      *
      * @param token Access token used to verify the caller's right to
perform this operation on this DMS.
      * @param status CommandStatus object used to track the progress of this
operation (may be null).
      *
      * @exception Common::AccessDenied Caller does not have proper rights to
perform this
      *                                  operation on this DMS.
      * @exception Common::CHART2Exception An error occurred while performing
this operation.
      */
      void pollNow(in Common::AccessToken token, in Common::CommandStatus
status)
         raises (Common::AccessDenied, Common::CHART2Exception);


      /**
      * This operation is used to reset the DMS controller in the field.
This operation will
      * also cause the sign to be blanked.
      * This operation executes asynchronously.  It returns to the caller
after queuing the
      * command for execution.  A CommandStatus object must be used if the
caller wishes
      * to track the progress of the operation.  A DMSBlanked event will be
pushed through the
      * Event Service to indicate the sign was blanked.
      *
      * @param token Access token used to verify the caller's right to
perform this operation on this DMS.
      * @param status CommandStatus object used to track the progress of this
operation (may be null).
      *
      * @exception Common::AccessDenied Caller does not have proper rights to
perform this
      *                                  operation on this DMS.
      * @exception ResourceManagement::ResourceControlConflict Used to
indicate that the resource is already
      *                                                          in use by
another op center.
      * @exception Common::CHART2Exception An error occurred while performing
this operation.
      */
      void resetController(in Common::AccessToken token, in
Common::CommandStatus status)
         raises (Common::AccessDenied,
ResourceManagement::ResourceControlConflict,
                 Common::CHART2Exception);


      /**
```

```
      * This operation returns the timeout used by the DMS to determine if
host communications
      * are present.  This value is configurable on some DMS models, while
fixed on others.
      * Some DMS models will automatically blank the sign if no comms to the
sign occur within
      * this timeout period.
      *
      * @return Comm loss timeout in minutes. (0 - 65535)
      */
      long getCommLossTimeout();


      /**
      * This operation sets the timeout used by the DMS to determine if host
communications
      * are present.  This operation is only valid on DMS models that support
this feature.
      * Setting this value lower than the polling interval may cause the sign
to blank its
      * display due to non-communications.
      * This operation executes asynchronously.  It returns to the caller
after queuing the
      * command for execution.  A CommandStatus object must be used if the
caller wishes
      * to track the progress of the operation.
      *
      * @param token Access token used to verify the caller's right to
perform this operation on this DMS.
      * @param timeout Timeout value in minutes. (0 - 65535)
      * @param status CommandStatus object used to track the progress of this
operation (may be null).
      *
      * @exception Common::AccessDenied Caller does not have proper rights to
perform this
      *                                  operation on this DMS.
      * @exception UnsupportedOperation This DMS does not support the setting
of the comm loss timeout.
      * @exception Common::CHART2Exception An error occurred while performing
this operation.
      */
      void setCommLossTimeout(in Common::AccessToken token, in long
dmsTimeCommLoss, in Common::CommandStatus status)
         raises (Common::AccessDenied, UnsupportedOperation,
Common::CHART2Exception);


      /**
      * This operation returns the size of the DMS.
      *
      * @return The size of the display area (in characters)
      */
      SignMetrics getSignMetrics();

      /**
```

```
      * This operation returns data regarding the size of the font used for
      * this DMS.  For this implemenatation, only one font is supported per
      * DMS and this font must be fixed width.
      */
      FontMetrics getFontMetrics();


      /**
      * This operation returns the operational status of the DMS.
      *
      * @return The operational status of this DMS.
      */
      DeviceManagement::OperationalStatus getOperationalStatus();


      /**
      * This operation returns the name of the network site where this DMS is
served.
      *
      * @return the name of the network site where the DMS is served.
      */
      Common::NetworkConnectionSite getNetConnectionSite();


      /**
      * This operation is used to create a plan item that involves setting a
particular
      * message on this DMS.  The plan item can then be stored as part of a
plan.
      *
      * @param token Access token used to verify the caller's right to
perform this operation on this DMS.
      * @param message Stored message being associated with this sign via a
plan item.
      *
      * @exception Common::AccessDenied Caller does not have proper rights to
perform this
      *                                  operation on this DMS.
      * @exception Common::CHART2Exception An error occurred while performing
this operation.
      *
      * @see Plan
      * @see PlanItem
      */
      DMSStoredMessageItem createPlanItem(in Common::AccessToken token,
                                          in StoredDMSMessage message)
         raises (Common::AccessDenied, Common::CHART2Exception);


      /**
      * This operation is used to remove this DMS from the system.
      *
      * @param token Access token used to verify the caller's right to
perform this operation on this DMS.
      *
```

```
      * @exception Common::AccessDenied Caller does not have proper rights to
perform this
      *                                   operation on this DMS.
      * @exception Common::CHART2Exception An error occurred while performing
this operation.
      */
      void remove(in Common::AccessToken token)
         raises(Common::AccessDenied, Common::CHART2Exception);
   };


   /**
   * This interface defines an object that is used to manage the creation and
deletion
   * of DMS objects in the system.
   */
   interface DMSFactory : ResourceManagement::SharedResourceManager
   {
      /**
      * This operation creates a DMS object within the factory.
      *
      * @param token Access token used to verify the caller's right to
perform this operation on this object.
      * @param config Configuration data for the DMS being created.
      *
      * @exception Common::AccessDenied Caller does not have proper rights to
perform this
      *                                   operation.
      * @exception Common::CHART2Exception An error occurred while performing
this operation.
      */
      DMS createDMS(in Common::AccessToken token, in Configuration config)
         raises(Common::AccessDenied, Common::CHART2Exception);
   };


   /**
   * This interface defines an object used to store a message that may be
placed on a DMS.
   */
   interface StoredDMSMessage
   {
      /**
      * This operation returns the unique identifier assigned to this object.
      *
      * @return the unique identifier assigned to this object.
      */
      Common::Identifier getID();


      /**
      * This operation returns the description of this stored message.
      *
      * @return The description of this stored message.
      */
```

```
      string getMessageDescription();


      /**
      * This operation sets the description of this stored dms message.
      *
      * @param token Access token used to verify the caller's right to
perform this operation on this object.
      * @param description Description of this stored message.
      *
      * @exception Common::AccessDenied Caller does not have proper rights to
perform this
      *                                  operation.
      * @exception Common::CHART2Exception An error occurred while performing
this operation.
      */
      void setMessageDescription(in Common::AccessToken token, in string
description)
         raises(Common::AccessDenied, Common::CHART2Exception);


      /**
      * This operation returns the message content stored in this stored
message.
      *
      * @return Message content stored in this stored message.
      */
      StoredMessageContent getMessageContent();


      /**
      * The operation sets the content of this stored message.
      *
      * @param token Access token used to verify the caller's right to
perform this operation on this object.
      * @param msgContent Content of message to be stored in this stored
message.
      *
      * @exception Common::AccessDenied Caller does not have proper rights to
perform this
      *                                  operation.
      * @exception DisapprovedMessageContent Message contains banned words or
the beacon state is invalid.
      * @exception Common::CHART2Exception An error occurred while performing
this operation.
      */
      void setMessageContent(in Common::AccessToken token, in
StoredMessageContent msgContent)
         raises(Common::AccessDenied, DisapprovedMessageContent,
Common::CHART2Exception);


      /**
      * This operation returns the minimum character width of a sign that
wishes to display this
```

```
      * stored message in a legible fashion.
      *
      * @return The minumum number of characters wide a DMS must be to
display the contents of this message.
      */
      long getMinCharacters();


      /**
      * This operation removes this stored message from the system.
      *
      * @param token Access token used to verify the caller's right to
perform this operation on this object.
      *
      * @exception Common::AccessDenied Caller does not have proper rights to
perform this
      *                                  operation.
      * @exception Common::CHART2Exception An error occurred while performing
this operation.
      */
      void remove(in Common::AccessToken token)
         raises(Common::AccessDenied, Common::CHART2Exception);
   };


   /**
   * This typedef defines a sequence of stored messages.
   */
   typedef sequence <StoredDMSMessage> StoredDMSMessageList;


   /**
   * This struct defines the data needed to create a StoredDMSMessage.
   *
   * @member msgContent Content of the message.
   * @member msgDescription Description of the message.
   * @member category Category assigned to this message.
   * @member minCharacters Minimum number of characters wide a DMS must be to
display this message.
   * @member createdBy Name of the user that created this message.
   */
   struct StoredDMSMessageCreationStruct
   {
      StoredMessageContent msgContent;
      string               msgDescription;
      string               category;
      long                 minCharacters;
      string               createdBy;
   };


   /**
   * This interface defines an object used as a collection of
StoredDMSMessages.
   *
```

```
   */
   interface DMSMessageLibrary
   {
      /**
      * This operation returns the unique identifier assigned to this object.
      *
      * @return The unique identifier assigned to this object.
      */
      Common::Identifier getID();


      /**
      * This operation returns the name of this library
      *
      * @return The name of this library.
      */
      string getName();


      /**
      * This operation sets the name of this library.
      *
      * @param token Access token used to verify the caller's right to
perform this operation on this object.
      * @param name Name to be assigned to this library.
      *
      * @exception Common::AccessDenied Caller does not have proper rights to
perform this
      *                                    operation.
      * @exception Common::CHART2Exception An error occurred while performing
this operation.
      */
      void setName(in Common::AccessToken token, in string name)
         raises(Common::AccessDenied, Common::CHART2Exception);


      /**
      * This operation creates a new StoredDMSMessage in this library.
      *
      * @param token Access token used to verify the caller's right to
perform this operation on this object.
      * @param message Message data used to create new StoredDMSMessage
      *
      * @return Newly created StoredDMSMessage object.
      *
      * @exception Common::AccessDenied Caller does not have proper rights to
perform this
      *                                                  operation.
      * @exception DisapprovedMessageContent Message content specified has
banned words or beacon state is invalid.
      * @exception Common::CHART2Exception An error occurred while performing
this operation.
      */
      StoredDMSMessage addMessage(in Common::AccessToken token, in
StoredDMSMessageCreationStruct message)
```

```
         raises(Common::AccessDenied, DisapprovedMessageContent,
Common::CHART2Exception);


      /**
      * This operation removes a StoredDMSMessage from this library and the
system.
      *
      * @param token Access token used to verify the caller's right to
perform this operation on this object.
      * @param message StoredDMSMessage to be removed.
      *
      * @exception Common::AccessDenied Caller does not have proper rights to
perform this
      *                                    operation.
      * @exception Common::CHART2Exception An error occurred while performing
this operation.
      */
      void removeMessage(in Common::AccessToken token, in StoredDMSMessage
message)
         raises(Common::AccessDenied, Common::CHART2Exception);


      /**
      * This operation returns a list of the stored messages contained in
this library.
      *
      * @return List of stored messages contained in this library.
      */
      StoredDMSMessageList getStoredMessages();


      /**
      * This operation returns a list of plans that are using a given stored
message.
      *
      * @return List of plans using this stored message.  The returned list
will be
      *         empty if there are no plans using the message.
      */
      PlanManagement::PlanList getPlansUsingMessage(in StoredDMSMessage msg);


      /**
      * This operation removes the library from the system.
      *
      * @param token Access token used to verify the caller's right to
perform this operation on this object.
      *
      * @exception Common::AccessDenied Caller does not have proper rights to
perform this
      *                                    operation.
      * @exception Common::CHART2Exception An error occurred while performing
this operation.
      */
```

```
      void remove(in Common::AccessToken token)
         raises(Common::AccessDenied, Common::CHART2Exception);
   };


   /**
   * This typedef defines a list of DMSMessageLibrary objects.
   */
   typedef sequence <DMSMessageLibrary> DMSMessageLibraryList;


   /**
   * This interface defines a factory used for creating and removing
DMSMessageLibraries.
   */
   interface DMSLibraryFactory
   {
      /**
      * This operation creates a new library.
      *
      * @param token Access token used to verify the caller's right to
perform this operation on this object.
      * @param name Name of the library to be created.
      * @return Newly created DMSMessageLibrary object.
      * @exception Common::AccessDenied Caller does not have proper rights to
perform this
      *                                  operation.
      * @exception Common::CHART2Exception An error occurred while performing
this operation.
      */
      DMSMessageLibrary createLibrary(in Common::AccessToken token, in string
name)
         raises(Common::AccessDenied, Common::CHART2Exception);


      /**
      * This operation returns a list of Libraries that are managed by this
factory.
      *
      * @return A List of libraries being managed by this factory.
      */
      DMSMessageLibraryList getLibraryList();
   };


   /**
   * This interface defines a plan item that is used to associate a DMS with
a
   * stored message.  When a plan item of this type is activated it will call
   * setMessage on the DMS passing it the message content to display.
   */
   interface DMSStoredMessageItem : PlanManagement::PlanItem
   {
      /**
      * This operation sets the DMS contained in this plan item.
```

```
      *
      * @param token Access token used to verify the caller's right to
perform this operation on this object.
      * @param dms DMS where message will be set when this plan item is
activated.
      * @exception Common::AccessDenied Caller does not have proper rights to
perform this
      *                                    operation.
      * @exception Common::CHART2Exception An error occurred while performing
this operation.
      *
      * @see DMS
      * @see StoredDMSMessage
      */
      void setDMS(in Common::AccessToken token, in DMS dms)
         raises(Common::AccessDenied, Common::CHART2Exception);


      /**
      * This method returns the DMS that will receive the message if this
plan item is
      * activated.
      *
      * @return the DMS that will receive the message if this plan item is
activated.
      */
      DMS getDMS();


      /**
      * This operation sets the Stored Message contained in this plan item.
      *
      * @param token Access token used to verify the caller's right to
perform this operation on this object.
      * @param message Message to be set on the DMS when this plan item is
activated.
      *
      * @exception Common::AccessDenied Caller does not have proper rights to
perform this
      *                                    operation.
      * @exception Common::CHART2Exception An error occurred while performing
this operation.
      */
      void setMessage(in Common::AccessToken token, in StoredDMSMessage
message)
         raises(Common::AccessDenied, Common::CHART2Exception);


      /**
      * This operation returns the message to be set on the DMS when this
plan item is activated.
      *
      * @return the message to be set on the DMS when this plan item is
activated.
      */
```

```
      StoredDMSMessage getMessage();
   };
};

#endif



/*
File Name      :  PlanManagement.idl

Prepared by    :  Computer Sciences Corporation / PB Farradyne, Inc.

+------+   Copyright 1999, Maryland Department of Transportation
| MDOT |                   All Rights Reserved
+------+-----------------------------------------------------------
This file is property of the Maryland Department of
Transportation.  Any unauthorized use or duplication of this
file, or removal of the header, constitutes theft of
intellectual property.

Description    :  This file contains the IDL definitions for plans.

History
Date        Author                  Description
-------------------------------------------------------------------------------
-
*/


#ifndef _PLAN_MANAGEMENT_IDL_
#define _PLAN_MANAGEMENT_IDL_

#include <Common.idl>

#pragma prefix "CHART2"


/**
* This module contains the definitions of the IDL interfaces
* used in creating plans.  Plans represent a task which is
* repetative in nature that the system will automate.  Each
* plan consists of a set of items and may be activated.
* When the plan is activated, it will tell each item to activate
* It is up to the item to determine what actions need to be taken
* in order to perform activation.
*/
module PlanManagement
{
   interface PlanItem;
   interface Plan;

   /**
   * A collection of plan items
   */
```

```
   typedef sequence<PlanItem> PlanItemList;

   /**
   * A collection of plans
   */
   typedef sequence<Plan> PlanList;

   /**
   * This enum describes the types of events that can be
   * pushed for plans.  When a plan item is added or modified
   * it is up to the derived item type to push the appropriate
   * event.
   *
   * @member PlanAdded Pushed when a plan is added.
   * @member PlanRemoved Pushed when a plan is removed.
   * @member PlanItemRemoved Pushed when a plan item is removed.
   *
   * @see PlanEvent
   */
   enum PlanEventType
   {
      PlanAdded,
      PlanRemoved,
      PlanItemRemoved
   };


   /**
   * This union maps the event types for plans to the
   * data necessary for those events.
   *
   * @see PlanEventType
   */
   union PlanEvent switch (PlanEventType)
   {
      case PlanAdded:
         Plan plan;

      case PlanRemoved:
         Common::Identifier planID;

      case PlanItemRemoved:
         Common::Identifier planItemID;
   };


   /**
   * Interface for the base plan item.  Each plan item is capable of being
   * activated.  This interface will serve as the base for future items which
   * each know the specific actions which need to be performed upon
activation.
   *
   * @see Plan
   * @see CHART2.DMSControl.DMSStoredMessageItem
   */
```

```
   interface PlanItem
   {
      /**
      * Returns the unique identifier of the plan item.
      *
      * @return The unique identifier of the plan item.
      */
      Common::Identifier getID();


      /**
      * Changes the name of the plan item.
      *
      * @param token - Access token used to verify the caller's right to
      *                perform this operation.
      * @param name - New name of the plan item.
      *
      * @exception Common::AccessDenied - Caller does not have proper
      *                                   rights to perform this operation.
      * @exception Common::CHART2Exception - A general error occurred while
      *                                      attempting the command.
      */
      void setName(in Common::AccessToken token, in string name)
         raises (Common::AccessDenied, Common::CHART2Exception);


      /**
      * Returns the name of the plan item.
      *
      * @return The name of the plan item.
      */
      string getName();


      /**
      * Activates the plan item.
      *
      * @param token - Access token used to verify the caller's right to
perform this
      *                operation.
      * @param commandStatus - **Optional** Object which will be called to
show the
      *                        progress of the command if the client wants
status
      *                        updates.  Pass null if status updates are not
      *                        needed.
      *
      * @exception Common::AccessDenied - Caller does not have proper
      *                                   rights to perform this operation.
      * @exception Common::CHART2Exception - A general error occurred while
      *                                      attempting the command.
      */
      void activate(in Common::AccessToken token,
                    in Common::CommandStatus commandStatus)
         raises (Common::AccessDenied, Common::CHART2Exception);
```

```
      /**
      * Removes the plan item from the system.
      *
      * @param token - Access token used to verify the caller's right to
      *                perform this operation.
      *
      * @exception Common::AccessDenied - Caller does not have proper
      *                                   rights to perform this operation.
      * @exception Common::CHART2Exception - A general error occurred while
      *                                      attempting the command.
      */
      void remove(in Common::AccessToken token)
         raises (Common::AccessDenied, Common::CHART2Exception);


      /**
      * Determines whether the given object is being used by the plan item.
      *
      * @param objectID - The ID of the object to check the plan item for.
      *
      * @return True if the plan item is using the object, false otherwise.
      */
      boolean isUsingObject(in Common::Identifier objectID);
   };


   /**
   * Interface for a plan.  The plan contains zero or more plan items.  A
plan
   * can be activated, at which time it will activate each of the plan items
   * that it contains.
   *
   * @see PlanItem
   */
   interface Plan
   {
      /**
      * Returns the unique identifier of the plan.
      *
      * @return - The unique identifier of the plan.
      */
      Common::Identifier getID();


      /**
      * Changes the name of the plan.
      *
      * @param token  - Access token used to verify the caller's right to
      *                 perform this operation.
      * @param name - New name of the plan.
      *
      * @exception Common::AccessDenied - Caller does not have proper
      *                                   rights to perform this operation.
```

```
      * @exception Common::CHART2Exception - A general error occurred while
      *                                      attempting the command.
      */
      void setName(in Common::AccessToken token, in string name)
         raises (Common::AccessDenied, Common::CHART2Exception);


      /**
      * Returns the name of the plan.
      *
      * @return The name of the plan.
      */
      string getName();


      /**
      * Adds a plan item to the plan.
      *
      * @param token  - Access token used to verify the caller's right to
perform
      *                 this operation.
      * @param item  - Item to add to the plan.
      *
      * @exception Common::AccessDenied - Caller does not have proper
      *                                   rights to perform this operation.
      * @exception Common::CHART2Exception - A general error occurred while
attempting
      *                                      the command.
      */
      void addItem(in Common::AccessToken token, in PlanItem item)
         raises (Common::AccessDenied, Common::CHART2Exception);


      /**
      * Removes an item from the plan.
      *
      * @param token - Access token used to verify the caller's right to
perform
      *                this operation.
      * @param item - Plan item to remove from the plan.
      *
      * @exception Common::AccessDenied - Caller does not have proper
      *                                   rights to perform this operation.
      * @exception Common::CHART2Exception - A general error occurred while
      *                                      attempting the command.
      */
      void removeItem(in Common::AccessToken token, in PlanItem item)
         raises (Common::AccessDenied, Common::CHART2Exception);


      /**
      * Attempts to activate all of the plan items.
      *
      * It is the responsibility of the plan, when activated, to summarize
      * the status of the activation of each plan item and report that
```

```
      * information to the CommandStatus object if one is passed.
      *
      * @param token - Access token used to verify the caller's right to
perform
      *                this operation.
      * @param commandStatus - **Optional** Will be called as the individual
      *                        plan items are being activated to show status.
      *                        Pass null if status updates are not needed.
      *
      * @exception Common::AccessDenied - Caller does not have proper
      *                                   rights to perform this operation.
      * @exception Common::CHART2Exception - A general error occurred while
      *                                      attempting the command.
      */
      void activate(in Common::AccessToken token, in Common::CommandStatus
commandStatus)
         raises (Common::AccessDenied, Common::CHART2Exception);


      /**
      * Returns all of the plan items in the plan.
      *
      * @return All of the plan items in the plan.
      */
      PlanItemList getItems();


      /**
      * Determines whether the given object is being used by the plan.
      *
      * @param objectID - The ID of the object to check the plan for.
      *
      * @return - True if the plan is using the object, false otherwise.
      */
      boolean isUsingObject(in Common::Identifier objectID);
   };


   /**
   * This interface manages all of the system plans.  It creates each one
   * and stores it in a collection.  It has methods which operate on all of
the
   * plans as a group.
   */
   interface PlanFactory
   {
      /**
      * Creates a new plan and adds it to the system.
      *
      * @param token - Access token used to verify the caller's right to
perform
      *                this operation.
      * @param name - The name of the new plan.
      *
      * @exception Common::AccessDenied - Caller does not have proper
```

```
      *                                    rights to perform this operation.
      * @exception Common::CHART2Exception - A general error occurred while
      *                                      attempting the command.
      */
      Plan createPlan(in Common::AccessToken token, in string name)
         raises (Common::AccessDenied, Common::CHART2Exception);


      /**
      * Returns all of the plans in the factory.
      *
      * @returns All of the plans in the factory.
      */
      PlanList getPlans();


      /**
      * Determines which of the plans are using the object having the
      * specified identifier.
      *
      * @param objectID Identifier of the object to query for.
      *
      * @returns All plans in the factory which are currently using the
object.
      */
      PlanList getPlansUsingObject(in Common::Identifier objectID);
   };

};  // end module PlanManagement

#endif



/*
File Name      :  ResourceManagement.idl

Prepared by    :  Computer Sciences Corporation / PB Farradyne, Inc.

+------+   Copyright 1999, Maryland Department of Transportation
| MDOT |                   All Rights Reserved
+------+-----------------------------------------------------------
This file is property of the Maryland Department of
Transportation.  Any unauthorized use or duplication of this
file, or removal of the header, constitutes theft of
intellectual property.

Description    :  This file contains IDL definitions for the Resource
Management
                  module.  This includes the interfaces:
                     SharedResource
                     SharedResourceManager
                     OperationsCenter
                     UserLoginSession
                     Organization
```

```
History
Date        Author                  Description
---------------------------------------------------------------------------
-

*/

#ifndef _RESOURCE_MANAGEMENT_IDL_
#define _RESOURCE_MANAGEMENT_IDL_

#include <Common.idl>

#pragma prefix "CHART2"


/**
* This module contains the definitions of the IDL interfaces
* pertaining to shared resources, operations centers, user login sessions,
* and organizations.
*/
module ResourceManagement
{
   interface SharedResource;
   interface UserLoginSession;


   /**
   * A sequence of shared resources.
   */
   typedef sequence<SharedResource>       SharedResourceList;


   /**
   * A sequence of user login sessions.
   */
   typedef sequence<UserLoginSession>     LoginSessionList;


   /**
   * Exception which describes a login failure.
   *
   * @member reason - the explanation for the failure.
   */
   exception LoginFailure
   {
      string reason;
   };


   /**
   * Exception which describes a logout failure.
   *
   * @member reason - the explanation for the failure.
   */
```

```
   exception LogoutFailure
   {
      string reason;
   };


   /**
   * Exception which describes a failure caused when the user tries
   * to do something which requires that no resources be controlled, yet
   * the Operations Center in which the user is logged in to is
   * still controlling one or more shared resources.
   *
   * @member reason - the explanation for the failure.
   */
   exception HasControlledResources
   {
      string reason;
   };


   /**
   * Exception which describes a failure caused when the user attempts
   * to do something with the resource, but the resource is already being
   * controlled by another operations center.
   *
   * @member reason - the explanation for the failure.
   */
   exception ResourceControlConflict
   {
      string reason;
   };


   // ------------------------------------------------------------

   /**
   * Enumeration which describes all of the resource-related event types.
   *
   * @member UnhandledControlledResourcesEvent - event pushed when an
OperationsCenter has controlled
   *                                            resources but no logged in
users.
   *
   * @see ResourceEvent
   */
   enum ResourceEventType
   {
      UnhandledControlledResourcesEvent
   };


   /**
   * Event data pushed when an OperationsCenter has controlled
   * resources but no logged in users.
   *
```

```
   * @member opCtrID - the ID of the Operations Center with controlled
resources
   * @member opCtrName - the name of the Operations Center
   */
   struct UnhandledControlledResourcesInfo
   {
      Common::Identifier opCtrID;
      string opCtrName;
   };


   /**
   * Union which maps all resource-related event types to the
   * associated event data.
   *
   * @member unhandledResources - data describing the
UnhandledControlledResourcesEvent
   *                             event type.
   *
   * @see ResourceEventType
   */
   union ResourceEvent switch(ResourceEventType)
   {
      case UnhandledControlledResourcesEvent:
         UnhandledControlledResourcesInfo unhandledResources;
   };


   // ---------------------------------------------------------

   /**
   * Interface which represents an Operations Center. It contains
   * functionality for logging users in and out and managing controlled
resources.
   */
   interface OperationsCenter
   {
      /**
      * Returns the unique identifier of the operations center.
      *
      * @return the unique identifier of the operations center
      */
      Common::Identifier getID();


      /**
      * Returns the name of the operations center.
      *
      * @return the name of the operations center
      */
      string getName();


      /**
      * Logs a user into the operations center.  If successful, it will
```

```
      * generate an access token which the caller must store for the user
      * to perform any restricted operations.
      *
      * @param loginSession - the login session to keep track of.
      * @param userName - the user's login ID.
      * @param password - the user's password.
      *
      * @return The access token which is required for restricted CHART2
operations.
      *
      * @exception LoginFailure indicates a login failure, possibly caused by
an
      *                          invalid user name or password.
      */
      Common::AccessToken loginUser(in UserLoginSession loginSession,
                           in Common::UserName userName,
                           in string password)
         raises (LoginFailure);


      /**
      * Logs a user out of the operations center.  This requires
      * that no controlled resources be assigned to the operations
      * center if this is the last person to log out.
      *
      * @param loginSession - the login session to log out.
      *
      * @exception LogoutFailure indicates a logout failure, possibly caused
by
      *                            an invalid login session
      * @exception HasControlledResources indicates that this is the last
person
      *                                    to log out of the Op Center, yet
there
      *                                    are still shared resources being
controlled.
      */
      void logoutUser(in UserLoginSession loginSession)
         raises (LogoutFailure, HasControlledResources);


      /**
      * Gets the shared resources which are currently being
      * controlled by this operations center.
      *
      * @return The shared resources which are currently being controlled by
      *         this operations center.
      */
      SharedResourceList getControlledResources();


      /**
      * Gets all of the users currently logged into this operations center.
      *
      * @param token - access token required for the restricted operation
```

```
      *
      * @return The login session objects for all logged in users.
      *
      * @exception Common::AccessDenied indicates that the caller does not
have
      *                                  permission to call this method.
      */
      LoginSessionList getLoginSessions(in Common::AccessToken token)
         raises (Common::AccessDenied);


      /**
      * Forces the specified user to be logged out from the system.
      * This action will not be prevented if there are controlled resources
      * and it is the last user to be logged out.
      *
      * @param token - access token required for the restricted operation.
      * @param loginSession - the login session of the user to be logged out.
      *
      * @exception Common::AccessDenied indicates that the caller does not
have
      *                           permission to call this method.
      * @exception LogoutFailure indicates that the logout failed, possibly
due
      *                            to an unknown or invalid login session.
      */
      void forceLogout(in Common::AccessToken token, in UserLoginSession
loginSession)
         raises (Common::AccessDenied, LogoutFailure);


      /**
      * Gets the number of users currently logged in to this operations
center.
      *
      * @return The number of users currently logged in.
      */
      long getNumLoggedInUsers();


      /**
      * Transfers control of the shared resources to the target operations
center.
      *
      * @param token - access token required for the restricted operation.
      * @param resources - the shared resources to transfer control of.
      * @param targetOpCenter - the operations center to transfer control to.
      *
      * @exception Common::AccessDenied indicates that the caller does not
have
      *                           permission to call this method.
      */
      void transferSharedResources(in Common::AccessToken token,
                                   in SharedResourceList resources,
                                   in OperationsCenter targetOpCenter)
```

```
         raises (Common::AccessDenied);

   };  // end interface OperationsCenter


   // ------------------------------------------------

   /**
   * Interface which represents an instance of a user being logged in at
   * an Operations Center.
   */
   interface UserLoginSession
   {
      /**
      * Gets the operations center that the user is logged in to
      *
      * @return The login session objects for all logged in users.
      */
      OperationsCenter getOpCenter();


      /**
      * Gets the login ID of the user.
      *
      * @return The login ID of the user.
      */
      Common::UserName getUsername();


      /**
      * Tests to make sure the LoginSession still exists
      * Note - if it does not exist, a CORBA exception will
      *        be thrown, so the return statement will not be executed.
      *
      * @return True if the LoginSession still exists.
      */
      boolean ping();


      /**
      * Forces the logout of the user.  This will do clean up
      * for the user being logged out.
      *
      * @param token - access token required for the restricted operation
      *
      * @exception Common::AccessDenied indicates that the caller does not
have
      *                          permission to call this method.
      */
      void forceLogout(in Common::AccessToken token)
         raises (Common::AccessDenied);

   };  // end interface UserLoginSession
```

```
   // ---------------------------------------------------------

   /**
   * Interface which represents an Organization.
   */
   interface Organization
   {
      /**
      * Returns the unique identifier of the organization.
      *
      * @return The unique identifier of the organization
      */
      Common::Identifier getID();


      /**
      * Returns the name of the organization.
      *
      * @return The name of the organization
      */
      string getName();
   };


   // ---------------------------------------------------------

   /**
   * Interface used by all shared resources which can be controlled
   * by an operations center.
   */
   interface SharedResource
   {
      /**
      * Returns the unique identifier of the shared resource.
      *
      * @return The unique identifier of the shared resource.
      */
      Common::Identifier getID();


      /**
      * Returns the operations center which currently has control of this
resource, or null.
      *
      * @return The operations center which currently has control of this
resource,
      *         or null if no operations center currently controls the
resource.
      */
      OperationsCenter getControllingOpCenter();


      /**
      * Returns the organization which owns the resource.
```

```
      *
      * @return The organization which owns the resource.
      */
      Organization getOwnerOrg();


      /**
      * Returns true if the resource is currently being controlled
      * by the given operations center.
      *
      * @param opCtrID - the operations center to check
      * @return True if the resource is controlled by the operations center;
otherwise, false.
      */
      boolean isControlledBy(in Common::Identifier opCtrID);


      /**
      * Sets the controlling operations center to be the specified operations
center.
      *
      * @param token - access token required for the restricted operation
      * @param opCIDtr - the operations center to set as controlling
      * @exception Common::AccessDenied indicates that the caller does not
have
      *                          permission to call this method.
      */
      void setControllingOpCenter(in Common::AccessToken token, in
Common::Identifier opCtrID)
         raises (Common::AccessDenied);


      /**
      * Clears the controlling operations center.
      *
      * @param token - access token required for the restricted operation
      * @exception Common::AccessDenied indicates that the caller does not
have
      *                          permission to call this method.
      */
      void clearControllingOpCenter(in Common::AccessToken token)
         raises (Common::AccessDenied);

   };  // end interface SharedResource


   // ----------------------------------------------------------

   /**
   * This interface maintains a collection of zero or more shared resources,
   * and has methods for dealing with them as a group.
   */
   interface SharedResourceManager
   {
      /**
```

```
      * Gets all of the shared resources in the manager.
      *
      * @return All of the shared resources in the manager.
      */
      SharedResourceList getResources();


      /**
      * Returns the shared resources which are currently being controlled by
the
      * given operations center.
      *
      * @param opCtr - the specified controlling operations center
      *
      * @return The shared resources which are currently being controlled by
the
      *         given operations center.
      */
      SharedResourceList getControlledResources(in OperationsCenter opCtr);


      /**
      * Determines whether there are any shared resources which are being
      * controlled by the given operations center.
      *
      * @param opCtr - the specified controlling operations center.
      * @return True if any of the shared resources are currently being
controlled
      *         by the given operations center.
      */
      boolean hasControlledResources(in OperationsCenter opCtr);

   };  // end interface SharedResourceManager

};  // end module ResourceManagement

#endif



/*

File Name    : UserManagement.idl

Prepared By  : Computer Sciences Corporation / PB Farradyne, Inc.

+------+   Copyright 1999, Maryland Department of Transportation
| MDOT |                   All Rights Reserved
+------+-----------------------------------------------------------
This file is property of the Maryland Department of
Transportation.  Any unauthorized use or duplication of this
file, or removal of the header, constitutes theft of
intellectual property.

Description  : This file contains interface definition language desribing
```

```
               the module containing access control related definitions for
               the CHART 2 project.

History
Date        Author         Description
--------------------------------------------------------------------------------
-

*/

#ifndef _CHART2_USERMANAGEMENT_IDL_
#define _CHART2_USERMANAGEMENT_IDL_

#include <Common.idl>


#pragma prefix "CHART2"

/**
* Interfaces needed to configure user profiles.
*
* This module contains the interfaces necessary to mange and utilize user
* profiles.
*/
module UserManagement
{
   /**
   * Name assigned to a role.
   *
   * The role name must be unique and must be no longer than
   * 32 bytes.
   */
   typedef string RoleName;

   /**
   * Represents a set of user capabilities.
   *
   * Each role has a name which uniquely identifies it and
   * a verbose description.  The description is used to
   * allow a system administrator to attach a meaningful
   * description to the role as a reminder of what it means to
   * grant it to a user.  Each role is assigned zero or more functional
   * rights.  Roles are then assigned to users to grant them
   * access to system functions.
   *
   * @member name - Name of this role
   * @member description - Verbose description of the role
   */
   struct Role
   {
      RoleName name;
      string description;
   };
```

```
/**
* Represents a particular user capability.
*
* A functional right grants a particular capability to perform
* system functions.  Each functional right may be limited by
* attaching the identifier of a particular organization to which
* this right is constrained.  This capability allows an administrator
* to grant a particular Role the ability to modify only shared
* resources owned by the identified organization.  The orgFilter
* identifier CHART2 will allow access to any organizations shared
* resources.
*
* @member id -        Unique identifier for this system capability
* @member orgFilter - The id of an organization whose shared
*                     resources this functional right is
*                     granting access.
*
* @see CHART2.ResourceManagement.SharedResource
*/
struct FunctionalRight
{
   unsigned long id;
   Common::Identifier orgFilter;
};


/**
* A list of functional rights
*/
typedef sequence<FunctionalRight> FunctionalRightList;

/**
* A list of roles
*/
typedef sequence<Role> RoleList;

/**
* A list of user names
*/
typedef sequence<Common::UserName> UserList;


//UserManager exceptions

/**
* The username specified is not valid.
*
* @member name - The invalid name.
* @member reason - Reason the name is invalid.
*/
exception InvalidUserName
{
   Common::UserName name;
   string reason;
};
```

```
/**
* The password specified is invalid.
*/
exception InvalidPassword
{
   Common::Password password; //The invalid password
   string reason;             //Reason the password is not valid
};

/**
* Thrown when an attempt is made to define a role which already
* exists.
*/
exception DuplicateRole
{
};

/**
* Thrown when the specified role name does not exist in the
* database.
*/
exception InvalidRole
{
};

/**
* Thrown when an attempt is made to delete a role which has
* users assigned to it.
*/
exception RoleInUse
{
};

/**
* Thrown when an attempt is made to add an invalid functional right
* to a role.
*
* @member right - The invalid right.
* @member reason - Reason the right is not valid.
*/
exception InvalidFunctionalRight
{
   FunctionalRight right;
   string reason;
};

/**
* Thrown when a user name is passed that is not in the user database.
*/
exception UnknownUser
{
};

/**
```

```
   * Thrown when the password specified for a user does not
   * match that user's password in the database.
   */
   exception IncorrectPassword
   {
   };

   /**
   * Thrown when an attempt is made to delete a user who is currently
   * logged in.
   */
   exception UserLoggedIn
   {
   };


   /**
   * Interface for configuration of user accounts for the CHART2 system.
   *
   * This interface provides access to information in the CHART2 system user
   * database.  It allows for the complete configuration of user accounts and
   * querying of a users current information.
   */
   interface UserManager
   {
      /**
      * Creates a new user.
      *
      * This method will create a new user with the specified user name and
password in the
      * user database.
      *
      * @param token - Access token of the client attempting to create the
user.
      * @param username - Name of new user to add.
      * @param password - Password of new user.
      *
      * @exception Common::AccessDenied - Thrown when the client does not
      *                                   have the appropriate functional
      *                                   rights to access this method.
      * @exception InvalidUserName - Thrown when the specified user name is a
duplicate
      *                              of an existing user or otherwise
violates user
      *                              naming conventions for the CHART2
system.
      * @exception InvalidPassword - Thrown when the specified password
violates
      *                              password conventions for the CHART2
system.
      * @exception Common::CHART2Exception - Thrown when a general error
occurs.
      */
      void createUser(in Common::AccessToken token, in Common::UserName
username, in Common::Password password)
```

```
         raises(Common::AccessDenied, InvalidUserName, InvalidPassword,
Common::CHART2Exception);


      /**
      * Deletes an existing user.
      *
      * This method will delete the specified user from the system database.
      *
      * @param token - Access token of the client attempting to delete the
user.
      * @param username - Name of new user to delete.
      *
      * @exception Common::AccessDenied - Thrown when the client does not
      *                                   have the appropriate functional
      *                                   rights to access this method.
      * @exception UnknownUser - Thrown when the specified user does not
exist in
      *                          the system user database.
      * @exception UserLoggedIn - Thrown if the user is currently logged in.
      * @exception Common::CHART2Exception - Thrown when a general error
occurs.
      */
      void deleteUser(in Common::AccessToken token, in Common::UserName
username)
         raises(Common::AccessDenied, UnknownUser, UserLoggedIn,
Common::CHART2Exception);


      /**
      * Gets a list of all users in the system.
      *
      * This method will return a list of all users in the system user
      * database.
      *
      * @param token - Access token of the invoking client.
      *
      * @return - The current list of user names in the database.
      *
      * @exception Common::AccessDenied - Thrown when the client does not
      *                                   have the appropriate functional
      *                                   rights to access this method.
      * @exception Common::CHART2Exception - Thrown when a general error
occurs.
      */
      UserList getUsers(in Common::AccessToken token)
         raises (Common::AccessDenied, Common::CHART2Exception);


      /**
      * Creates a new role.
      *
      * This method will create a new role in the system user database.
      *
      * @param token - Access token of the invoking client.
```

```
      *
      * @exception Common::AccessDenied - Thrown when the client does not
      *                                   have the appropriate functional
      *                                   rights to access this method.
      * @exception DuplicateRole - Thrown when the name of the role to create
      *                            matches the name of an existing role in
the
      *                            system user database.
      * @exception Common::CHART2Exception - Thrown when a general error
occurs.
      */
      void createRole(in Common::AccessToken token, in Role role)
         raises(Common::AccessDenied, DuplicateRole,
Common::CHART2Exception);


      /**
      * Gets a list of all roles in the system.
      *
      * This method will return a list of all roles in the system user
      * database.
      *
      * @param token - Access token of the invoking client.
      *
      * @return - The current list of roles in the database.
      *
      * @exception Common::AccessDenied - Thrown when the client does not
      *                                   have the appropriate functional
      *                                   rights to access this method.
      * @exception Common::CHART2Exception - Thrown when a general error
occurs.
      */
      RoleList getRoles(in Common::AccessToken token)
         raises (Common::AccessDenied, Common::CHART2Exception);


      /**
      * Grants the specified role to the specified user.
      *
      * This method will grant the specified role to the specified user.
      * this action will result in the user having all functional rights
      * assigned to the role upon his/her next login.
      *
      * @param token - Access token of the invoking client.
      * @param user - Name of the user to grant the role to.
      * @param role - Name of the role to grant.
      *
      * @exception Common::AccessDenied - Thrown when the client does not
      *                                   have the appropriate functional
      *                                   rights to access this method.
      * @exception DuplicateRole - Thrown when the user already has
previously been
      *                            granted the specified role.
      * @exception InvalidRole - Thrown when the role does not exist in the
user
```

```
      *                           database.
      * @exception UnknownUser - Thrown when the user does not exist in the
user
      *                           database.
      * @exception Common::CHART2Exception - Thrown when a general error
occurs.
      */
      void grantRole(in Common::AccessToken token, in Common::UserName user,
in RoleName role)
         raises(Common::AccessDenied, DuplicateRole, InvalidRole,
UnknownUser, Common::CHART2Exception);


      /**
      * Revokes the specified role from the specified user.
      *
      * This method will revoke the specified role from the specified user.
      * This action will result in the user having reduced functional rights
      * upon his/her next login.
      *
      * @param token - Access token of the invoking client.
      * @param user - Name of the user to revoke the role from.
      * @param role - Name of the role to revoke.
      *
      * @exception Common::AccessDenied - Thrown when the client does not
      *                                   have the appropriate functional
      *                                   rights to access this method.
      * @exception InvalidRole - Thrown when the role does not exist in the
user
      *                           database.
      * @exception UnknownUser - Thrown when the user does not exist in the
user
      *                           database.
      * @exception Common::CHART2Exception - Thrown when a general error
occurs.
      */
      void revokeRole(in Common::AccessToken token, in Common::UserName user,
in RoleName role)
         raises(Common::AccessDenied, InvalidRole, UnknownUser,
Common::CHART2Exception);


      /**
      * Deletes the specified role from the system user database.
      *
      * This method will delete the specified role from the system user db.
      * This method will not succeed if any users in the database are
currently
      * assigned to the role.
      *
      * @param token - Access token of the invoking client.
      * @param role - Name of the role to delete.
      *
      * @exception Common::AccessDenied - Thrown when the client does not
      *                                   have the appropriate functional
```

```
      *                                    rights to access this method.
      * @exception InvalidRole - Thrown when the role does not exist in the
user
      *                              database.
      * @exception Common::CHART2Exception - Thrown when a general error
occurs.
      */
      void deleteRole(in Common::AccessToken token, in RoleName role)
         raises(Common::AccessDenied, InvalidRole, RoleInUse,
Common::CHART2Exception);


      /**
      * Gets the functional rights currently assigned to a role.
      *
      * This method will get the list of functional rights currently
      * assigned to the specified role.
      *
      * @param token - Access token of the invoking client.
      * @param role - Name of the role to get the rights of.
      *
      * @return - The functional rights currently assigned to the
      *           specified role.
      *
      * @exception Common::AccessDenied - Thrown when the client does not
      *                                   have the appropriate functional
      *                                   rights to access this method.
      * @exception InvalidRole - Thrown when the role does not exist in the
user
      *                              database.
      * @exception Common::CHART2Exception - Thrown when a general error
occurs.
      */
      FunctionalRightList getRoleFunctionalRights(in Common::AccessToken
token, in RoleName role)
         raises(Common::AccessDenied, InvalidRole, Common::CHART2Exception);


      /**
      * Sets the functional rights currently assigned to a role.
      *
      * This method will set the list of functional rights currently
      * assigned to the specified role.  After this method completes,
      * the role will contain only the rights specified by this method.
      * The changes to the role will not take effect until the next time
      * a user who has this role assigned logs in.
      *
      * @param token - Access token of the invoking client.
      * @param role - Name of the role to get the rights of.
      *
      * @exception Common::AccessDenied - Thrown when the client does not
      *                                   have the appropriate functional
      *                                   rights to access this method.
      * @exception InvalidRole - Thrown when the role does not exist in the
user
```

```
      *                            database.
      * @exception Common::CHART2Exception - Thrown when a general error
occurs.
      */
      void setRoleFunctionalRights(in Common::AccessToken token, in RoleName
role, in FunctionalRightList rights)
         raises(Common::AccessDenied, InvalidRole, InvalidFunctionalRight,
Common::CHART2Exception);


      /**
      * Gets the roles a user may perform.
      *
      * This method will get the names of all roles that this user has
      * been granted from the system user database.
      *
      * @param token - Access token of the invoking client.
      * @param user - Name of the user to get the roles of.
      *
      * @return - The current list of roles that the specified user may
      *           play.
      *
      * @exception Common::AccessDenied - Thrown when the client does not
      *                                   have the appropriate functional
      *                                   rights to access this method.
      * @exception UnknownUser - Thrown when the user does not exist in the
user
      *                            database.
      * @exception Common::CHART2Exception - Thrown when a general error
occurs.
      */
      RoleList getUserRoles(in Common::AccessToken token, in Common::UserName
user)
         raises (Common::AccessDenied, UnknownUser, Common::CHART2Exception);


      /**
      * Sets the roles a user may perform.
      *
      * This method will set the list of roles that a user may perform.
      * This method differs from grantRole and revokeRole in that after it
      * completes, the user will be assigned only those roles which were
      * specified in the roles parameter.  The user will not obtain
      * his/her new role assignments until the next login.
      *
      * @param token - Access token of the invoking client.
      * @param user - Name of the user to get the roles of.
      * @param roles - List of all roles this user may play.
      *
      * @exception Common::AccessDenied - Thrown when the client does not
      *                                   have the appropriate functional
      *                                   rights to access this method.
      * @exception UnknownUser - Thrown when the user does not exist in the
user
      *                            database.
```

```
      * @exception InvalidRole - Trhown when a role in the list of roles is
not
      *                          in the system user database.
      * @exception Common::CHART2Exception - Thrown when a general error
occurs.
      */
      void setUserRoles(in Common::AccessToken token, in Common::UserName
user, in RoleList roles)
         raises (Common::AccessDenied, UnknownUser, InvalidRole,
Common::CHART2Exception);


      /**
      * Changes the password for a user.
      *
      * This method will change a user's system password.  The oldPassword
      * parameter must match the user's current system password.
      *
      * @param token - Access token of the invoking client.
      * @param user - Name of the user to change the password of.
      * @param oldPassword - Current password for the specified user.
      * @param newPassword - New password for the specified user.
      *
      * @exception Common::AccessDenied - Thrown when the client does not
      *                                   have the appropriate functional
      *                                   rights to access this method.
      *
      * @exception UnknownUser - Thrown when the user does not exist in the
user
      *                          database.
      *
      * @exception IncorrectPassword - Thrown when the old password passed
does not
      *                               match the current password for the
user.
      *
      * @exception InvalidPassword - Thrown when the new password specified
is not
      *                              valid.
      *
      * @exception Common::CHART2Exception - Thrown when a general error
occurs.
      */
      void changeUserPassword(in Common::AccessToken token, in
Common::UserName name,
                              in Common::Password oldPassword, in
Common::Password newPassword)
         raises(Common::AccessDenied, UnknownUser, IncorrectPassword,
InvalidPassword,
                Common::CHART2Exception);


      /**
      * Sets the password for a user.
      *
```

```
      * This method will change a user's system password.  This version
requires
      * the invoking client to have special functional rights because the
user's
      * current password is not needed.
      *
      * @param token - Access token of the invoking client.
      * @param user - Name of the user to change the password of.
      * @param newPassword - New password for the specified user.
      *
      * @exception Common::AccessDenied - Thrown when the client does not
      *                                   have the appropriate functional
      *                                   rights to access this method.
      *
      * @exception UnknownUser - Thrown when the user does not exist in the
user
      *                          database.
      *
      * @exception InvalidPassword - Thrown when the new password specified
is not
      *                              valid.
      *
      * @exception Common::CHART2Exception - Thrown when a general error
occurs.
      */
      void setUserPassword(in Common::AccessToken token, in Common::UserName
name,
                           in Common::Password new_pw)
         raises(Common::AccessDenied, UnknownUser, InvalidPassword,
Common::CHART2Exception);

   };

};

#endif
```